the magazine of the accu

www.accu.org

{cvu}

Volume 27 • Issue 5 • November 2015 • £3

Features

Bug Hunting
Pete Goodliffe

One Definition Rule
Roger Orr

Building C & C++ CLI Programs with libCLImate
Matthew Wilson

Functional Programming in C++
Richard Falconer

Raspberry Pi Linux User Mode GPIO in C++
Ralph McArdell

Regulars

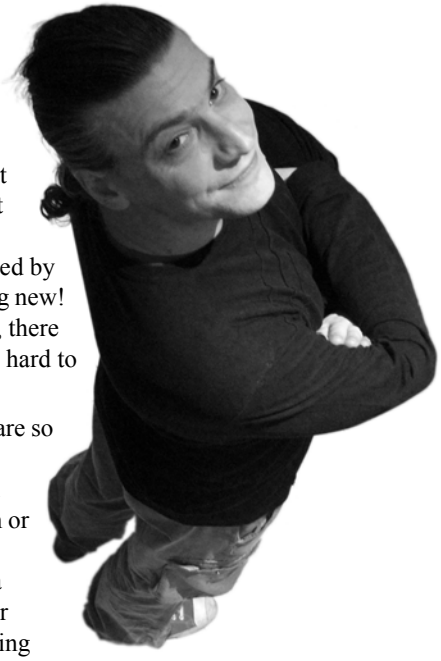
C++ Standards Report

Robert Martin: An Interview

Code Critique

{cvu}
Volume 27 Issue 5
November 2015
ISSN 1354-3164
www.accu.org

Selective ignorance



I was reading the blog of one of the lead developers of an open-source library recently, and it was interesting to note some of the remarks made about code quality and user friendliness. The user friendliness of an API is about how other developers perceive it, and of course it's a subjective thing – to a point. Code quality is one of the factors in any attempt to improve user friendliness, so if the quality is low, it can be very hard to change in order to improve it. Nothing remarkable about that, it's an issue experienced by lots of programmers every day, and is certainly nothing new! And that's the problem, right there – it's nothing new, there are loads of libraries and programs that are needlessly hard to use.

Very often the problem arises because the developers are so fixated on the underlying structure of the technology they're exposing – be it some hardware interface with lots of registers, a security protocol, a database system or whatever – that they forget about what the user might actually want to do. How many times have you used a library that required you to invoke several functions or methods to initialize something, and pass arcane-looking parameters that had to be in the right combinations for anything to work? Of course, you can write your own wrapper to do the right thing, but **ALL** users of the API have to do it, and they'll all do it slightly differently, and not be able to take advantage of the commonality.

Scott Meyers made the point about making things (he was talking about class interfaces, but it applies here) easy to use correctly, and hard or impossible to use incorrectly. This requires a little bit of thought on the part of the API developer, in making sensible defaults, preventing incorrect combinations of flags and/or parameters, naming the public API artefacts sensibly, and so on. Some programming languages help more than others in this regard, in the support they provide for abstraction. And that's what this is really about – a higher level of abstraction. If the API doesn't operate at a high enough abstraction, it becomes a *dis*traction.

</rant>

STEVE LOVE
FEATURES EDITOR

The official magazine of ACCU

ACCU is an organisation of programmers who care about professionalism in programming. That is, we care about writing good code, and about writing it in a good way. We are dedicated to raising the standard of programming.

ACCU exists for programmers at all levels of experience, from students and trainees to experienced developers. As well as publishing magazines, we run a respected annual developers' conference, and provide targeted mentored developer projects.

The articles in this magazine have all been written by programmers, for programmers – and have been contributed free of charge.

To find out more about ACCU's activities, or to join the organisation and subscribe to this magazine, go to www.accu.org.

Membership costs are very low as this is a non-profit organisation.

DIALOGUE

19 Functional Programming in C++
Richard Falconer reports on an ACCU talk by Kevlin Henney.

21 Code Critique Competition
Competition 96 and the answer to 95.

27 Robert Martin: An Interview
Emyr Williams continues the series of interviews with people from the world of programming

28 Standards Report
Jonathan Wakely reports the latest on C++17 and beyond.

REGULARS

28 ACCU Members Zone
Membership news.

FEATURES

3 Bug Hunting
Pete Goodliffe looks for software faults.

5 Building C & C++ CLI Programs with the libCLImate Mini-framework
Matthew Wilson presents a framework for simplifying CLI programs.

11 Raspberry Pi Linux User Mode GPIO in C++ (Part 3)
Ralph McArdell demonstrates the library with two peripherals on the Pi.

16 One Definition Rule
Roger Orr explains an often misunderstood aspect of C++.

SUBMISSION DATES

C Vu 27.6 1st December 2015
C Vu 28.1: 1st February 2016

Overload 131: 1st January 2016
Overload 131: 1st March 2016

WRITE FOR C VU

Both C Vu and Overload rely on articles submitted by you, the readers. We need articles at all levels of software development experience. What are you working on right now? Let us know!

Send articles to cvu@accu.org. The friendly magazine production team is on hand if you need help or have any queries.

ADVERTISE WITH US

The ACCU magazines represent an effective, targeted advertising channel. 80% of our readers make purchasing decisions or recommend products for their organisations.

To advertise in the pages of C Vu or Overload, contact the advertising officer at ads@accu.org.

Our advertising rates are very reasonable, and we offer advertising discounts for corporate members.

COPYRIGHTS AND TRADE MARKS

Bug Hunting

Pete Goodliffe looks for software faults.

If debugging is the process of removing software bugs, then programming must be the process of putting them in.
~ Edsger Dijkstra

It's open season; a season that lasts all year round. There are no permits required, no restrictions levied. Grab yourself a shotgun and head out into the open software fields to root out those pesky varmints, the elusive *bugs*, and squash them, dead.

OK, reality is not as saccharin as that. But sometimes you end up working on code in which you swear the bugs are multiplying and ganging up on you. A shotgun is the only response.

The story is an old one, and it goes like this: Programmers write code. Programmers aren't perfect. The programmer's code isn't perfect. It therefore doesn't work perfectly the first time. So we have bugs.

If we bred better programmers, we'd clearly breed better bugs.

Some bugs are simple mistakes that are obvious to spot and easy to fix. When we encounter these, we are lucky.

The majority of bugs – the ones we invest hours of effort tracking down, losing our follicles and/or hair pigment in the search – are the nasty, subtle issues. These are the odd, surprising interactions; the unexpected consequences of our algorithms; the seemingly non-deterministic behaviour of software that looked so very simple. It can only have been infected by gremlins.

This isn't a problem limited to newbie programmers who don't know any better. Experts are just as prone. The pioneers of our craft suffered; the eminent computer scientist Maurice Wilkes [1] wrote:

> I well remember [...] on one of my journeys between the EDSAC room and the punching equipment that 'hesitating at the angles of stairs' the realisation came over me with full force that a good part of the remainder of my life was going to be spent in finding errors in my own programs.

So face it. You'll be doing a lot of debugging. You'd better get used to it. And you better get good at it. (At least you can console yourself that you'll have plenty of chances to practice.)

An economic concern

How much time do you think is spent debugging? Add up the effort of all of the programmers in every country around the world. Go on, guess.

A staggering $312 billion per year is spent on the wage bills for programmers debugging their software. To put that in perspective, that's two times all Eurozone bailouts since 2008! This huge, but realistic, figure comes from research carried out by Cambridge University's Judge Business School. [2]

You have a responsibility to fix bugs faster: *to save the global economy*. The state of the world is in your hands.

It's not just the wage bill, though. Consider all the other implications of buggy software: shipping delays, cancelled projects, the reputation damage from unreliable software, and the cost of bugs fixed in shipping software.

An ounce of prevention

It would be remiss of an article about debugging to not stress how much better it is to actively prevent bugs manifesting in the first place, rather than attempt a post-bug fix. *An ounce of prevention is worth a pound of cure.* If the cost of debugging is astronomical, we should primarily aim to mitigate this by not creating bugs in the first place.

This, in a classic editorial sleight of hand, is material for a different article, and so we won't investigate the theme exhaustively here. Do remember how important it is to expect the unexpected and to always work with your brain fully engaged!

Suffice to say, we should always employ sound engineering techniques that minimise the likelihood of unpleasant surprises. Thoughtful design, code review, pair programming, and a considered test strategy (including TDD practices and fully automated unit test suites) are all of the utmost importance. Techniques like assertions, defensive programming, and code coverage tools will all help minimise the likelihood of errors sneaking past.

We all know these mantras. Don't we? But how diligent are we in employing such tactics?

> Avoid injecting bugs into your code by employing sound engineering practices. Don't expect quickly hacked-out code to be of high quality.

The best bug-avoidance advice is to not write incredibly 'clever' (which often equates to complex) code. Brian Kernighan states: "Debugging is twice as hard as writing the code in the first place. Therefore, if you write the code as cleverly as possible, you are, by definition, not smart enough to debug it." Martin Fowler reminds us: "Any fool can write code that a computer can understand. Good programmers write code that humans can understand."

Bug hunting

Being realistic, no matter how sound your code-writing regimen, some of those pernicious bugs will always manage to squeeze through the defences. Donald Knuth once wrote: "Beware of bugs in the above code; I have only proved it correct, not tried it."

The programmer will always be required to don their hunting cap and an anti-bug shotgun.

How should we go about finding and eliminating bugs? This can be a Herculean task, akin to finding a needle in a haystack. Or, more accurately, a needle in a needle stack.

Finding and fixing a bug is like solving a logic puzzle. Generally the problem isn't too hard when approached methodically; the majority of bugs are easily found and fixed in minutes. However, some are nasty and take longer. Those hard bugs are few in number, but given their nature, that's where we will spend most of our time.

Two factors usually determine how hard a bug is to fix:

- How reproducible it is.
- The time between the cause of the bug entering the code, the 'software fault' itself – the bad line of code, or faulty integration assumption – and you actually noticing.

When a bug scores highly on both counts, it's almost impossible to track down without sharp tools and a keen intellect. There are a number of practical techniques and strategies we can employ to solve the puzzle and locate the fault.

PETE GOODLIFFE

Pete Goodliffe is a programmer who never stays at the same place in the software food chain. He has a passion for curry and doesn't wear shoes. Pete can be contacted at pete@goodliffe.net or @petegoodliffe

The first, and most important thing, is to methodically investigate and characterise the bug. Give yourself the best raw material to work with:

- Reduce it to the simplest set of *reproduction steps* possible. This is vital. Sift out all the extraneous fluff that isn't contributing to the problem, and only serves to distract.
- Ensure that you are focusing on a single problem. It can be very easy to get into a tangle when you don't realise you're conflating two separate – but related – faults into one.
- Determine how repeatable the problem is. How frequently do your repro steps demonstrate the problem? Is it reliant on a simple series of actions? Does it depend on software configuration or the type of machine you're running on? Do peripheral devices attached make any difference? These are all crucial data points in the investigation work that is to come.

In reality, when you've constructed a single set of reproduction steps, you really have won most of the battle.

So let's look at some of the most useful debugging strategies...

Lay traps

You have errant behaviour. You know a point when the system seems correct; maybe it's at start-up, but hopefully a lot later through the reproduction steps. You can get it to a point where its state is invalid. Find places in the code path *between* these two points, and set traps to catch the fault.

Add assertions or tests to verify the system *invariants* – the facts that must hold for the state to be correct.

Add diagnostic printouts to see the state of the code so you can work out what's going on.

As you do this, you'll gain a greater understanding of the code, reasoning more about its structure, and will likely add many more assertions to the mix to prove your assumptions hold. Some of these will be genuine assertions about invariant conditions in the code, others will be assertions relevant to this particular run. Both are valid tools to help you pinpoint the bug. Eventually a trap will snap, and you'll have the bug cornered.

> Assertions and logging (even the humble `printf`) are potent debugging tools. Use them often.

Diagnostic logs and assertions may be valid to leave in the code after you've found and fixed the problem. But be careful you don't litter the code with useless logging that hides what's really going on, making unnecessary debug noise.

Learn to binary chop

Aim for a *binary chop* strategy, to focus in on bugs as quickly as possible.

Rather than single-stepping through code paths, work out the start of a chain of events, and the end. Then partition the problem space into two, and work out if the middle point is good or bad. Based on this information, you've narrowed the problem space to something half the size. Repeat this a few times, and you'll soon have honed in on the problem.

This is a very powerful approach – allowing you to get to a solution in order *O(log n)* time, rather than *O(n)*. That is significantly faster.

> Binary chop problem spaces to get results faster.

Employ this technique with trap laying. Or with the other techniques described next.

Employ software archaeology

Software archaeology describes the art of mining through the historical records in your version control system. This can provide an excellent route into the problem; it's often a surprisingly simple way to hunt a bug.

Determine a point in the near past of the codebase when this bug didn't exist. Armed with your reproducible test case, step forward in time to determine which code changeset caused the breakage. Again, a binary chop strategy is the best bet here. (The `git bisect` tool automates this binary chop for you, and is worth keeping in your toolbox if you're a Git user.)

Once you find the breaking code change, the cause of the fault is usually obvious, and the fix is self-evident. (This is another compelling reason to make series of small, frequent, atomic check-ins, rather than massive commits covering a range of things at once.)

Test, test, test

As you develop your software, invest time to write a suite of unit tests. This will not only help shape how you develop and verify the code you've initially written. It acts as a great early warning device for changes you make later; much like the miner's canary, the test fails long before the problem becomes complex to find and expensive to fix.

These tests can also act as great points from which to begin debugging sessions. A simple, reproducible unit test case is a far simpler scaffold to debug than a fully running program that has to spin up and have a series of manual actions run to reproduce the fault. For this reason, it's advisable to write a unit test to demonstrate a bug, rather than start to hunt it from a running 'full system'.

Once you have a suite of tests, consider employing a *code coverage* tool to inspect how much of your code is actually covered by the tests. You may be surprised. A simple rule of thumb is: if your test suite does not exercise it, then you can't believe it works. Even if it looks like it's OK now, without a test harness then it'll be very likely to get broken later.

> Untested code is a breeding ground for bugs. Tests are your bleach.

When you finally determine the cause of a bug, consider writing a simple test that clearly illustrates the problem, and add it to the test suite *before* you really fix the code. This takes genuine discipline, as once you find the code culprit, you'll naturally want to fix it ASAP and publish the fix. Instead, first write a test harness to demonstrate the problem, and use this harness to prove that you've fixed it. The test will serve to prevent the bug coming back in the future.

Next time

These are by no means no the only debug strategies. In the next article we'll cover some other useful debugging strategies. Until then, may all your bugs be easy to find... ■

Questions

1. Assess how much of your time you think you spend debugging. Consider every activity that isn't writing a fresh line of code in a system.
2. Do you spend more time debugging new lines of code you have written, or on adjustments to existing code?
3. Does the existence of a suite of unit tests for existent code change the amount of time you spend debugging, or the way you debug?
4. What other bug-hunting strategies do you find valuable?
5. Is it realistic to aim for bug-free software? Is this achievable? When is it appropriate to genuinely aim for bug-free software? What determines the optimal amount of 'bugginess' in a product?

Reference

[1] Maurice Wilkes, *Memoirs of a Computer Pioneer* (Cambridge, MA: The MIT Press, 1985)

[2] 'Cambridge University Study States Software Bugs Cost Economy $312 Billion per Year' – http://undo-software.com/company/press/press-release-8

Building C & C++ CLI Programs with the libCLImate Mini-framework

Matthew Wilson presents a framework for simplifying CLI programs.

This article, the third in a series looking at software anatomy, builds on the material discussed in the first two instalments by discussing **libCLImate**, a mini-framework for command line interface (CLI) programming that encapsulates as much of the boilerplate as possible, and how it may be used in combination with program suite-specific libraries that encapsulate the rest, leaving the CLI application programmer to concentrate only on the interesting parts of the application development.

Introduction

In the first instalment of this series, 'Anatomy of a CLI Program written in C' [1], I considered in some depth the different aspects of CLI program construction, and expressed my desire to find a way to stop spending so much time thinking about and working on the fundamental aspects of program construction and focus instead on the interesting parts of the different problems programs need to solve. In the second instalment, 'Anatomy of a CLI Program written in C++' [2], I considered how to apply disparate utility libraries in the design and implementation of CLI programs, with the intention of separating out the boring boilerplate from the program-specific code, to promote flexibility, reuse, and testability.

In this, the third instalment, I discuss the reification of these previous intentions and considerations in the form of the **libCLImate** mini-framework, and its use in combination with program suite-specific libraries to drastically simplify the effort in creating CLI programs in C and C++.

libCLImate requirements

The requirements for the framework included:

- Require of the application programmer only a single entry-point function and declarative specification of command-line arguments;
- Support C and C++ without compromise to either;
- Handle (un)initialisation of all dependency libraries;
- Hide away as much boring boilerplate as possible (without detracting too much from discoverability);
- Make as much of the non-hideable boring boilerplate as possible be declarative;
- Facilitate strict separation of the *action logic* from the *decision logic*, *support logic*, and *declarative logic* to support the principle of ***program design is library design*** [1].

In essence: make the job of the CLI programmer less boring, so they can do it faster and better.

Listing 1

```
#include <libclimate/main/api.h>
int
main(
  int     argc
, char**  argv
)
{
  void* reserved = NULL;
  int (*pfn)(int, char**, void*);
#ifdef __cplusplus
  pfn = libCLImate_main_entry_point_Cpp;
#else /* ? __cplusplus */
  pfn = libCLImate_main_entry_point_C;
#endif /* __cplusplus */
  return (*pfn)(argc, argv, reserved);
}
```

Tour of the code

Because the **libCLImate** library comprises just a few small source files, it might be most illuminating to learn about the library by walking through the code.

Interface

There are six header files of interest to a user of the library (under the `include` directory):

- `libclimate/main.h`
- `libclimate/main.hpp`
- `libclimate/main/api.h`
- `libclimate/main/api.hpp`
- `libclimate/implicit_link/common_implicit_link.i`
- `libclimate/implicit_link/core.h`

libclimate/main.h

If you're writing a C program you would probably include `libclimate/main.h` (an elided form of which is shown in Listing 1).

As is strikingly obvious, including this file defines for you the **`main()`** entry point for the program, in terms of

- **`libCLImate_main_entry_point_Cpp`** (if C++), or
- **`libCLImate_main_entry_point_C`** (if C)

both of which we'll discuss shortly. Naturally, you can only include this file into one compilation unit of your program.

libclimate/main.hpp

If you're writing a C++ program you would probably include `libclimate/main.hpp`:

```
#include <libclimate/main.h>
#include <libclimate/main/api.hpp>
```

Note that I've said 'probably' in both cases. If you're using a framework that itself provides **`main()`** – I recall that ACE [3] does that, and there are doubtless others – then you would instead eschew these two files and go to the underlying API files `libclimate/main/api.h` and `libclimate/main/api.hpp`.

MATTHEW WILSON

Matthew is a software development consultant and trainer for Synesis Software who helps clients to build high-performance software that does not break, and an author of articles and books that attempt to do the same. He can be contacted at matthew@synesis.com.au.

libclimate/main/api.h

The first of these, `libclimate/main/api.h`, is the primary header file for the library. The abridged contents are shown in Listing 2.

Listing 2

```
#include <libclimate/common.h>
#include <libclimate/internal/clasp.clasp.h>
#include <stdio.h>

/* API globals */
/* The application-defined CLASP aliases array */
#ifdef __cplusplus
extern "C"
#else /* ? __cplusplus */
extern
#endif /* __cplusplus */
clasp_alias_t const libCLImate_aliases[];

/* API callbacks */
/* Application-defined program entry point
   (for C) */
#ifdef __cplusplus
extern "C"
#else /* ? __cplusplus */
extern
#endif /* __cplusplus */
int
libCLImate_program_main_C(
  clasp_arguments_t const* args
);

/* Application-defined program entry point
   (for C++) */
#ifdef __cplusplus
extern "C++"
int
libCLImate_program_main_Cpp(
  clasp_arguments_t const* args
);
# define libCLImate_program_main
libCLImate_program_main_Cpp
#else /* ? __cplusplus */
# define libCLImate_program_main
libCLImate_program_main_C
#endif /* __cplusplus */

/* API functions */
#ifdef __cplusplus
extern "C"
{
#endif

/* main */
#ifdef __cplusplus
int
libCLImate_main_entry_point_Cpp(
  int     argc
, char**  argv
, void*   reserved
);
#endif /* __cplusplus */

#ifndef __cplusplus
int
libCLImate_main_entry_point_C(
  int     argc
, char**  argv
, void*   reserved
);
#endif /* !__cplusplus */
```

Listing 2 (cont'd)

```
/* exit */
void
libCLImate_exit_immediately(
  int   exitCode
, void (*pfn)(int exitCode, void* param)
, void* param
) /* noexcept */
;
#ifdef __cplusplus
extern "C++"
void
libCLImate_unwind_and_exit(
  int   exitCode
);
#endif /* __cplusplus */
/* usage */
int
libCLImate_show_usage(
  clasp_arguments_t const*  args
, clasp_alias_t const*      aliases
, FILE*                     stm
, int                       verMajor
, int                       verMinor
, int                       verRevision
, int                       buildNumber
, char const*               programName
, char const*               summary
, char const*               copyright
, char const*               description
, char const*               usage
, int
showBlanksBetweenItems
);
int
libCLImate_show_usage_header(
  clasp_arguments_t const*  args
. . . // as libCLImate_show_usage()
);
int
libCLImate_show_usage_body(
  clasp_arguments_t const*  args
. . . // as libCLImate_show_usage()
);
int
libCLImate_show_version(
  clasp_arguments_t const*  args
, clasp_alias_t const*      aliases
, FILE*                     stm
, int                       verMajor
, int                       verMinor
, int                       verRevision
, int                       buildNumber
, char const*               programName
);

#ifdef __cplusplus
} /* extern "C" */
#endif
```

This breaks down as follows:

- application-defined constructs; and
- API functions:
  - the framework entry point(s);
  - early exit functions; and
  - usage helpers.

A bit grandiose, perhaps, but the **libCLImate** (mini-)framework can be considered a powerful and semi-intelligent **`ExecuteAroundMethod`** [4]. As such, the application programmer is required to provide the

Listing 3

```
/* example.c */
#include <libclimate/main.h>
#include <stdlib.h>
clasp_alias_t const libCLImate_aliases[] =
{
  CLASP_ALIAS_ARRAY_TERMINATOR
};
int
libCLImate_program_main_C(
  clasp_arguments_t const* args
)
{
  return EXIT_SUCCESS;
}
```

effective entry point and, so that command-line arguments can be processed on the application programmer's behalf (by the CLASP library [5]), the CLASP aliases array. Hence, there are three application-defined constructs, two of which must be defined (depending on whether you're writing a C or C++ program). The aliases array is familiar from both previous instalments in this series ([1, 2]) and for **libCLImate** it must have the name **`libCLImate_aliases`**. The effective entry point must be called **`libCLImate_program_main_C()`** (for C) or **`libCLImate_program_main_Cpp()`** (for C++), as in Listing 3.

There is a preprocessor macro **`libCLImate_program_main`** that resolves to **`libCLImate_program_main_Cpp`** or **`libCLImate_program_main_C`** as appropriate. You may wonder why not simply declare a single function of that name; this is explained later in this article.

The remainder of the contents of the main header file are API functions. The first pair are the framework entry points, as discussed earlier. If you are handling the definition of **`main()`** separate to **libCLImate** and so are not including `libclimate/main.h` or `libclimate/main.hpp`, then you will invoke one of these (**`libCLImate_main_entry_point_C()`** for C; **`libCLImate_main_entry_point_Cpp()`** for C++) once in your program's execution.

The early exit functions are next, and are discussed separately shortly.

The usage helpers are thin wrappers over the CLASP [5] facilities that have been discussed in previous instalments [1, 2], so I won't discuss them further here. Note, however, that they still involve many parameters – this will be addressed satisfactorily when I show how **libCLImate** may be used with CLI program suites.

libclimate/main/api.hpp

As you may have guessed from the earlier definition of `libclimate/main.hpp`, the contents of `libclimate/main/api.hpp` are defined largely in terms of `libclimate/main/api.h`:

```
#include <libclimate/main.h>
#include <libclimate/main/api.hpp>
```

libclimate/implicit_link/core.h

This is a standard-fare implicit link header file, for use with those compiler suites (such as Visual C++) that support that technique, to link implicitly to the requisite (to the compilation conditions, e.g. release, multithreaded, multibyte-string, ...) **libCLImate** static library.

libclimate/implicit_link/common_implicit_link.i

Since **libCLImate** is implemented in terms of several other libraries – **CLASP**, **Pantheios** [6], **recls** [7] (Windows-only), this file includes some common, always-used libraries' implicit-link headers (see Listing 4).

Pantheios aficionados will likely pick up that only the core and util libraries are specified, and why: front-end and back-end libraries are not specified, precisely because it is not the business of a general purpose mini-framework such as **libCLImate** to prescribe the specifics of the diagnostic logging control and output used by its users. Thus, if you're using implicit-linking with **libCLImate** you will need to specify additionally which front/back-end libraries you require; specifying linking of core and util will not be necessary (but is harmless if you do).

Listing 4

```
#include <libclimate/implicit_link/core.h>

#include <systemtools/clasp/implicit_link.h>

#include <pantheios/implicit_link/util.h>
#include <pantheios/implicit_link/core.h>

#ifdef _WIN32
# include <recls/implicit_link.h>
#endif
```

Implementation

I'm not going to show much of the implementation, as a lot of it is simply a gathering together of notions previously espoused (in this forum), and you're all welcome to simply browse it in (and fork it from!) the GitHub repo (http://github.com/synesissoftware/libCLImate).

There are five implementation files and one internal header file (in the `src` directory):

- `main_entry_point.c`
- `main_entry_point.cpp`
- `exit_immediately.c`
- `unwind_and_exit.cpp`
- `quiet_program_termination_exception.hpp`
- `usage_etc.c`

The first five of these are discussed in the next two sections. `usage_etc.c` does little more than provide the implementations for the helper functions mentioned earlier, including invoking the requisite CLASP API functions (using C Streams, aka **`FILE*`**) and determining the console width (if not piped).

Supporting C and C++

We all know that there can be only a single definition of **`main()`** in a link-unit. You are likely to know also that calling a C++ function that may throw exceptions from a C function yields undefined behaviour.

So, in order to support both C and C++ from the same library, we have to make sure that **`main()`** is not defined in the library. As you've already seen, **`main()`** is defined *within the program's object code* by including `libclimate/main.h(pp)` in one compilation unit in the project. It is then implemented in terms of either:

- **`libCLImate_main_entry_point_C()`** for C, which is defined in `main_entry_point.c`; or
- **`libCLImate_main_entry_point_Cpp()`** for C++, which is defined in `main_entry_point.cpp`.

These two files have the same logical structure:

- (un)initialise **Pantheios** with **Pantheios.Extras.Main** [8];
- trace outer-scope memory leaks with **Pantheios.Extras.DiagUtil** [9];
- (un)initialise CLASP.

However, the libraries used – **Pantheios.Extras.***, **CLASP** – have very different behaviours depending on whether they are compiled in C or C++. Most importantly, in C++: **Pantheios.Extras.Main** performs outermost-scope exception handling, issuing contingent reports (to standard error stream) and diagnostic log statements; and **CLASP** handles command line-related exceptions along the lines of "MyProgram: unrecognised flag '--stranger'; use --help for usage".

Thus, the separation is necessitated by the rules of the language as they pertain to **`main()`**'s uniqueness, by the need to support exceptions in order to provide rich handling of common non-normative conditions (such as a user specifying an unrecognised flag/option). That it also facilitates the

Listing 5

```
#include <stdio.h>
#include <stdlib.h>
class inout
{
public:
  inout() { fputs("in", stdout); }
  ~inout() { fputs(" & out\n", stdout); }
};
int main(int argc, char* argv[])
{
  inout io;
  if(1 != argc)
  {
    exit(1);
  }
  return 0;
}
```

ability of the user to provide his/her own **`main()`** and call into **libCLImate** explicitly may be thought a bonus.

Exceptions and early exit

The necessary handling of exceptions just mentioned also affords us the ability to take away another common (at least to me) but non-standard bit of repetitive work.

As you may know, gentle reader, when (**`std::`**)**`exit()`** is invoked, the implementation does not cause the destruction of any automatic variables. Hence, the output of Listing 5 is 'in and out' when run without arguments, but only 'in' when given one or more arguments.

What this means for sophisticated CLI programs is that calling (**`std::`**)**`exit()`** can be a bad idea, because things won't get cleaned up by the C++ runtime. For sure, many things, such as file handles, will be cleaned up by the operating system (and some things by the C runtime), but it may be unwise to rely on that, because none of it will be done with an understanding of what those objects were doing from a 'C++ point of view'. (I know that's a horribly woolly description, but I can't think of a good example right now, and I trust, gentle readers, that you can go with me regardless...)

Anyway, we do want to be able to perform an early exit to the program, and we don't (always) want to achieve this by *N* **`return`**s (where *N* can be a very large number). Well, in C++, there's a well-known control-transfer mechanism [10]: exceptions. Why not throw an exception?

Two problems. First, which exception do we throw? There's no standard exception to indicate a request to exit a program (or thread). I've written many such things over the years: some within general-purpose C++ libraries; others within program suites. The problem with the former is that it's more coupling for something totally fundamental and uninteresting. The problem with the latter is that one ends up in copy-paste hell. Coupling or copying – yuck!

The second problem is more subtle, but much more significant. It is received wisdom (which I too espouse [11]) that *all exception types should be derived from* **`std::exception`**. But really this rubric is too simplistic. What I think it should actually mean is that *all exceptions types* ***whose instances we may interact with*** *should be derived from* **`std::exception`**. The reason is that one should only catch what is one's business to catch. Or, put more powerfully, one should not catch what it is not one's business to catch.

If we define our putative *end-program-exception* to derive (by whatever depth) from **`std::exception`**, then it is possible that application code (or library code that is at a higher level of abstraction and dependency than our *end-program-exception*) may intercept and quench it. Of course, any code that does such a thing is (overwhelmingly likely) in error, but practical experience (in C++, and in many other languages – C# being the standout worst) tells me that this will happen.

So, there's a strong argument to be made to have an *end-program-exception* that is not part of the **`std::exception`** hierarchy. (Note: there are some arguments against, such as subversion of a **`std::exception`**-derivation assumption in the implementation of a C-API boundary, but I'm running out of space to discuss here. Pepper me with email on the accu-general mailing list if you wish.)

Going with this argument, however, means we risk exposing our exception to good practice – pun intended! – to the wider world, giving a bad example. That's where `quiet_program_termination_exception` comes in: it carries an exit code from the throw point, which is in **`libCLImate_unwind_and_exit()`** (in `unwind_and_exit.cpp`) to a handler in the **`ExecuteAroundMethod`** layered function stack in `main_entry_point_Cpp.cpp`. No user of **libCLImate** is exposed to this class, not even polymorphically.

Being practical, however, we must recognise that there may be some circumstances in which this isolation from **`std::exception`** is not desired. So, there are a bunch of pre-processor symbols that may be defined during the building of **libCLImate** that allow the exception to:

- have no inheritance (the default);
- inherit from **`std::exception`**;
- inherit from **`std::runtime_error`**; or
- inherit from **`stlsoft::unrecoverable`**, an STLSoft [12] exception type that may not be quenched, and if it is causes the program to exit.

The name

I asked the friendly fellows at ACCU-general for assistance in naming the library. There were several apposite suggestions (along with the odd inevitable 'wheel already invented' carp), but one stood out. Jonathan Wakeley offered **libCLImate**, ostensibly because it's a supporting (mate) library for CLI, but really so that Phil Nash could pun that 'every pull request will result in CLI-mate change'. In the face of such wit I was powerless to resist. ☺

Program suites

Applying **libCLImate** results in a substantial reduction in size and simplification of the implementation of standalone CLI programs. However, there are still some aspects that involve too much work:

- Since (almost) all CLI programs support the UNIX-standard flags **`--help`** and **`--version`**, having to detect and act on these flags, and in the same way, in every program is tiresome;
- Defining (and using) all 13 program identity attributes – version (major, minor, revision, build); strings (name, copyright, summary, description, usage) – is a drag, and leads to cluttered code (reducing transparency and increasing the likelihood of mistakes);
- **libCLImate**'s usage helpers are necessarily verbose, taking between 8 and 13 parameters;
- **libCLImate** does not select diagnostics transport (in the form of Pantheios back-ends) or control (in the form of Pantheios front-end), because it can't know what's appropriate for arbitrary CLI programs. Since (in my experience) program suites use the same diagnostics facilities in each program, having to do the same specifications and customisations of such facilities repeatedly is tiresome and duplicative, and may lead to copy-paste errors and/or divergence;
- Some program suites have common, but suite-specific, exceptions that may be caught and handled in a suite-common manner. Having to write this same code in each program is tiresome and duplicative, and may lead to copy-paste errors and/or divergence.

By combining **libCLImate** with a program suite-specific library, the implementation of each program in the suite can be distilled down so much as to achieve the primary expressed aim of the library: Require of the application programmer only a single entry-point function and declarative

specification of command-line arguments. In the remainder of this instalment I will demonstrate how that's been achieved for the Synesis Software Source Tools program suite via the proprietary library **libSrcToolMain**.

libSrcToolMain consists of three groups of source files:

- CLI framework files;
- CRT extension files; and
- Common utilities files.

We're interested in the CLI framework files, of which there are nine worth exploring:

- `include/SynesisSoftware/SourceTools/implicit_link/core.h`
- `src/common/common_implicit_link.i`
- `src/CLI/diagnostics.c`
- `include/SynesisSoftware/SourceTools/program_identity_globals.h`
- `include/SynesisSoftware/SourceTools/standard_argument_helpers.h`
- `src/CLI/show_usage.c`
- `src/CLI/standard_argument_helpers.cpp`
- `src/CLI/main.cpp`
- `src/CLI/tool_main_outer.cpp`

The files `core.h` and `common_implicit_link.i` serve the same functions as their **libCLImate** analogues. But because all programs in the program suite have common diagnostics, the latter file also includes implicit link inclusions for Pantheios' **`fe.simple`** front-end and **`be.N`** multiplexing back-end (in conjunction with the **`console`**, **`file`** and **`WindowsDebugger`** concrete back-ends).

The file `diagnostics.c` contains the program suite-specific back-end customisations, a small part of which is shown in Listing 6.

The file `program_identity_globals.h` declares a bunch of global constants that collectively define the identity and version of the given program (see Listing 7). The definitions of these constants are provided in each program's `identity.cpp` file (defined as described in [2]).

The file `standard_argument_helpers.h` declares a number of helper functions that are useful when processing the command-line (Listing 8).

As can be seen plainly, these functions do not have a list of parameters as long as your arm: that's because each one is defined (in `show_usage.c` and `standard_argument_helpers.cpp`) in terms of the requisite **libCLImate** API functions passing the identity globals as appropriate. This leads to much more transparent (and maintainable) application code.

But the real gain is in the two files not yet mentioned: `main.cpp` and `tool_main_outer.cpp`. `main.cpp` (Listing 9 includes `libclimate/main.hpp` and defines the (**libCLImate**-perspective) effective entry point **`libCLImate_program_main()`** in terms of the declared function **`tool_main_outer()`**, which it calls after first checking for, and reacting to, the **`--help`** and **`--version`** UNIX-standard flags.

The function **`tool_main_outer()`** is defined in `tool_main_outer.cpp` (Listing 10) in terms of the declared (**libSrcToolMain**-perspective) effective entry point **`tool_main_inner()`** which it invokes within a try-catch whose catch clauses handle program suite-specific exceptions in a suite-common manner.

While that might seem a lot of code to present in an article, it's pretty small in terms of a library, and it allows *all* programs within the program suite to substantially reduce the amount of boilerplate, such that each program's `entry.cpp` (defined as described in [2]) consists solely of the **`libCLImate_aliases`** array definition and the effective program entry **`point tool_main_inner()`**, as shown in Listing 11 (extracted from the **fsrtax** source tool's `entry.cpp`). The result is maximised application-specific code : boilerplate ratio in each program's

Listing 6

```
/* application-defined callbacks */
PANTHEIOS_CALL(void)
pantheios_be_WindowsDebugger_getAppInit(
  int                               backEndId
, pan_be_WindowsDebugger_init_t*  init
) /* throw() */
{
  STLSOFT_SUPPRESS_UNUSED(backEndId);

  init->flags |=
    PANTHEIOS_BE_INIT_F_NO_PROCESS_ID;
  init->flags |= PANTHEIOS_BE_INIT_F_HIDE_DATE;
  init->flags |=
    PANTHEIOS_BE_INIT_F_HIGH_RESOLUTION;
}
. . .
```

Listing 7

```
#include <SynesisSoftware/SourceTools/common.h>

/* application-defined globals */
#ifdef __cplusplus

extern "C"  const int         sourcetoolVerMajor;
extern "C"  const int         sourcetoolVerMinor;
extern "C"  const int
sourcetoolVerRevision;
extern "C"  const int
sourcetoolBuildNumber;

extern "C"  char const* const sourcetoolToolName;
extern "C"  char const* const sourcetoolSummary;
extern "C"  char const*
  const sourcetoolCopyright;
extern "C"  char const* const
  sourcetoolDescription;
extern "C"  char const* const sourcetoolUsage;

#else /* ? __cplusplus */
extern      const int         sourcetoolVerMajor;
. . .

#endif /* __cplusplus */
```

Listing 8

```
int SS_SrcTools_show_usage(
  clasp_arguments_t const*  args
, clasp_alias_t const*      aliases
, FILE*                     stm
);
int SS_SrcTools_show_version(
  clasp_arguments_t const*  args
, clasp_alias_t const*      aliases
, FILE*                     stm
);
void SS_SrcTools_display_usage_and_ \
    rewind_if_requested(
  clasp_arguments_t const*  args
, char const*               flagLongForm
, FILE*                     stm
);
void SS_SrcTools_display_version_and_ \
    rewind_if_requested(
  clasp_arguments_t const*  args
, char const*               flagLongForm
, FILE*                     stm
);
```

Listing 9

```
#include <SynesisSoftware/SourceTools/
  program_identity_globals.h>
#include <SynesisSoftware/SourceTools/
  standard_argument_helpers.h>
#include <libclimate/main.hpp>

extern "C++"
int
tool_main_outer(
  clasp::arguments_t const* args
);
int
libCLImate_program_main(
  clasp::arguments_t const* args
)
{
  namespace ssst = SynesisSoftware::SourceTools;

  /* process standard flags */
  ssst::display_usage_and_unwind_if_requested(
    args, "--help", stdout);
  ssst::display_version_and_unwind_if_requested(
    args, "--version", stdout);

  /* process command-line */
  return tool_main_outer(args);
}
```

`entry.cpp`; all the program suite-specific boilerplate is in the program suite library, and all the CLI boilerplate is in **libCLImate**.

Summary

For all that frameworks are loathsome things, restricting freedom and constraining choice, they are often a necessary evil in programming, because sometimes the advantages outweigh the disadvantages. I believe **libCLImate** represents just such a net-positive balance, and have been using it for some time now to focus mostly on the interesting parts of my CLI projects.

Next time

In the next instalment (which may be a while because I'm starting a new job next week!), I want to move away from C and C++ for a bit, perhaps to something a bit more modern such as Go, or maybe flesh out the structure I've adapted for writing Ruby CLI programs of late. ■

Listing 10

```
extern "C++"
int
tool_main_inner(
  clasp::arguments_t const* args
);

int
tool_main_outer(
  clasp::arguments_t const* args
)
{
  try
  {
    return tool_main_inner(args);
  }
  catch(<ps-specific-exc1>& x)
  {
    . . .
  }
  . . .
}
```

Acknowledgements

Thanks to Garth Lancaster for review pointers, and to Steve Love for editorial patience.

References

[1] Anatomy of a CLI Program written in C, Matthew Wilson, *CVu* September 2012.
[2] Anatomy of a CLI Program written in C++, Matthew Wilson, *CVu* September 2015.
[3] https://en.wikipedia.org/wiki/Adaptive_Communication_Environment
[4] http://c2.com/cgi/wiki?ExecuteAroundMethod
[5] An Introduction to CLASP, part 1: C, Matthew Wilson, *CVu*, January 2012
[6] http://pantheios.org/; http://github.com/synesissoftware/Pantheios
[7] http://recls.org/; http://github.com/synesissoftware/recls
[8] http://pantheios.org/; http://github.com/synesissoftware/Pantheios.Extras.Main
[9] http://pantheios.org/; http://github.com/synesissoftware/Pantheios.Extras.DiagUtil
[10] Quality Matters 5: Exceptions: The Worst Form of Exception Handling, Apart From All the Others, Matthew Wilson, *Overload*, #98, August 2010
[11] Quality Matters 6: Exceptions For Practically-Unrecoverable Conditions, Matthew Wilson, Overload, #99, October 2010
[12] http://stlsoft.org/; http://github.com/synesissoftware/STLSoft-1.9

Listing 11

```
#include "fsrtax.hpp"
#include "identity.h"
#include <SynesisSoftware/SourceTools/
program_identity_globals.h>
. . .
extern "C"
clasp::alias_t const libCLImate_aliases[] =
{
  // standard flags
  SS_SRCTOOLS_STD_FLAG_help(),
  SS_SRCTOOLS_STD_FLAG_version(),

  // program logic
  CLASP_BIT_FLAG("-D", "--details",
    FSRTAX_F_SHOW_DETAILS,
    "displays all details"),
  . . .
  CLASP_ALIAS_ARRAY_TERMINATOR
};
extern "C++"
int
tool_main_inner(
  clasp::arguments_t const* args
)
{
  int flags = clasp::check_all_flags(args,
    libCLImate_aliases);
  . . .
  clasp::verify_all_options_used(args);
  . . .
  return process(flags, . . .);
}
```

Raspberry Pi Linux User Mode GPIO in C++ (Part 3)

Ralph McArdell demonstrates the library with two peripherals on the Pi.

The previous instalments [1, 2] have described creating the **rpi-peripherals** [3] library to access general purpose input output (GPIO) on a Raspberry Pi running Raspbian Linux in C++ from user space. They covered creating the **phymem_ptr** class template that utilises RAII (resource acquisition is initialisation [4]) to manage mapped areas of physical memory, setting up the library project and the implementation of support for basic general purpose input and output of single bit Boolean values, clocks and pulse with modulation (PWM).

This final instalment describes the two serial interface peripheral types the library supports and closes with some concluding remarks.

SPI game

Having added PWM support to the library and played with the Gertboard's [5] motor controller I turned to the digital to analogue converter (DAC) and analogue to digital converter (ADC) chips. The ADC handles 10-bit samples while the DAC handles 8-bit samples – other Gertboards may have similar DAC chips that handle 10 or 12 bit samples. Both chips use the serial peripheral interface (SPI) [6] to transfer data making the addition of SPI support the next task.

The Raspberry Pi's Broadcom BCM2835 processor has several peripherals that support SPI. For the older Raspberry Pi models I was targeting the GPIO pins provided for interfacing only allow using the SPI0 peripheral – referred to as 'SPI' or 'SPI Master' in the documentation [7]. As there are also two instances of a 'Universal SPI Master' or 'mini SPI interface' peripheral type named SPI1 and SPI2 I decided to unambiguously use 'spi0' to refer to SPI0 peripheral related entities.

SPI uses the common master/slave model [8] where a single master device has control over one or more slave devices. In this case SPI0 on the Raspberry Pi is the master device and the ADC and DAC chips are slave devices. As an aside the BCM2835 also supports an SPI slave peripheral, unavailable on the earlier Raspberry Pi models. The standard communication mode is a '3-wire' protocol, allowing master and slave to send data simultaneously. Two of the three 'wires' are for the master to send data to a slave – MOSI (master output, slave input), and for the slave to send data to the master – MISO (master input, slave output). SPI is a synchronous serial interface so the third 'wire', SCLK, is the clock signal which is provided by the master. The master only communicates with one slave device at a time, which device being determined by a number of chip enable (or chip select) lines. The SPI0 peripheral provides two such lines – CE0 and CE1.

For added fun SPI0 supports two 2-wire modes which only use SCLK and MOSI for communication: bidirectional mode which calls MOSI MOMI (master output, master input) and LoSSI (low speed serial interface) mode which names MOSI SDA (serial data) and SCLK SCL (serial clock).

The data transferred between SPI0 and a slave device are a pair of FIFO (first in first out) buffers, one for data to be sent to the slave and the other for data received from the slave. They are 16 32-bit words deep, allowing 64 8-bit bytes to be buffered. The FIFOs are a case where peripheral registers do not behave like regular memory. Both FIFOs are accessed via the same register with reads returning the next available value from the receive FIFO and writes writing to the next available slot in the transmit FIFO.

As with other peripherals I only considered support for SPI0 polled use. On the other hand I did want to support all three communication modes along with various parameters that may need tweaking depending on the specifics of various slave devices.

A plurality of pins

Implementing SPI0 support initially followed the pattern discussed in part 2: I started with the **spi0_registers** class that matched the layout of the SPI0 peripheral's control registers and allows querying and setting the various fields. I created the **spi0_ctrl** singleton class which as per the pattern contains the **phymem_ptr** specialisation **phymem_ptr<volatile spi0_registers>** member that maps a **spi0_registers** instance over the SPI0 control registers. As SPI0 only supports a single item the type for the **spi0_ctrl** is-in-use tracking member is simply a **bool**.

The pattern dictates there should be a **spi0_pin** class. SPI0 requires the use of 5 pins, or 4 for the 2-wire modes. Hence the name **spi0_pins** – plural, was chosen.

It seemed cumbersome to pass in four or five pin value parameters to the **spi0_pins** constructor, especially considering that for the Raspberry Pi models I was targeting there could only be two possible valid sets of values: either all five of the available pins for SPI0 functions or the four pin subset required for 2 wire bidirectional mode operation. On the other hand I did not want to hard code the pin values as the SPI0 functions are available on a second group of pins and it was possible other BCM2835 based systems could access them.

So the constructor of **spi0_pins** takes a **spi0_pin_set** class template specialisation. **spi0_pin_set** has five integer non-type template parameters – one for each SPI0 pin. The fifth parameter has a default value representing pin-not-used to allow four pin sets to be defined. The class contains no values and only has five member functions – one for each SPI0 pin function – that return the value of the associated template parameter. Two instances are defined: **rpi_p1_spi0_full_pin_set** that specifies all 5 pins and **rpi_p1_spi0_2_wire_only_pin_set** that specifies the 4 pin subset required for 2-wire mode operation.

Chip chat

I soon realised that having just a **spi0_pins** class was not sufficient. Adding functionality to **spi0_pins** to allow checking the various states the FIFOs could be in was not a problem. It was when I came to implement the read and write functionality that I ran into a problem.

SPI0 can directly select one of two devices to converse with and switch conversations between devices. Each device can use different sets of communication parameters, so one **spi0_pins** instance has be able to support multiple sets of conversation parameters.

RALPH MCARDELL

Ralph McArdell has been programming for more than 30 years with around 20 spent as a freelance developer predominantly in C++. He does not ever want or expect to stop learning or improving his skills.

My initial solution was to create a `spi0_conversation` class, instances of which represented conversations with specific devices with the relevant parameters being passed as constructor arguments. To hold a conversation an instance was opened by passing a reference to a `spi0_pins` object to the `open` member function. Once successfully opened the conversation could proceed by calling the `read` and `write` member functions. When done the `close` member function was called. Attempting to have more than one open conversation at a time caused an exception to be thrown. Opening a conversation would set the required SPI0 communication parameters, then activate SPI0 data transfers – which were deactivated on conversation close.

This scheme worked but I did not like the low level twiddling with SPI0 registers being split across two classes. Worse, in order to keep track of open conversations the `spi0_pins` instance had to keep a pointer to the current open `spi0_conversation` which was set to `nullptr` on conversation close meaning the `spi0_conversation` object had to hold a pointer to the `spi0_pins` object. So instances of each type referenced each other whenever there was an open conversation which seemed somewhat inelegant.

Moving the read and write functionality to `spi0_pins` solved my first gripe. To address the second gripe I applied conversation contexts to a `spi0_pins` instance, with member functions `spi0_pins::start_conversing` and `spi0_pins::stop_conversing` in place of open and close operations.

This left `spi0_conversation` with very little to do so I renamed it `spi0_slave_context`. Instances of `spi0_slave_context` hold the subset of SPI0 peripheral registers needed to define the context of each conversation. To create the required register values the construction parameters are used to set the relevant field values of a local automatic `spi0_registers` object from which the required complete register values are copied to instance data members.

To start a conversation a `spi0_slave_context` instance is passed to `spi0_pins::start_conversing` which, after stopping any conversation and performing some validation, copies over most of the `spi0_slave_context` object's register value members to their live counterparts. Some fields of the control and status register should not be overwritten so a mask is used to apply only the relevant bits.

Read and write à la mode

The read and write operations provided by `spi0_pins` cater for transferring single and multiple byte values. Single byte operations return a `bool` indicating whether or not a value was transferred to or from a FIFO. Multi-byte operations return a `std::size_t` indicating how many bytes were transferred. A transfer may not complete for several reasons: conversing may be stopped or the transmit FIFO is full or the receive FIFO empty.

Which of the three SPI communication modes is used is part of a conversation's associated `spi0_slave_context`. The specifics of the protocols used by each mode are mostly hidden behind the `read` and `write` member functions, although some modes require extra information for some operations. Luckily these could be supported by appending an additional parameter with a default value to the operations concerned.

The easiest mode to implement, and the only one I could fully test due to the available hardware, was standard 3-wire mode. In standard mode data is written to the transmit FIFO while it is not full and read from the receive FIFO while it is not empty. The only oddity is that in order to read some data you must first write something – anything. So to read two bytes you first write two bytes. As these are transmitted to the slave device its reply is received, appearing in the receive FIFO. Of course serially transferring data takes time so there will be delays involved.

Listing 1 shows an example where a conversation expects two bytes to be received after the slave device has been sent a single byte that sets up the conversation. In the `read` function, while only one byte is required to be written to setup the conversation the same value is sent twice as we expect to read two bytes. Each byte is then read by calling the `read_byte` function which waits until data arrives in the receive FIFO, wrapping a call to the single byte overload of `spi0_pins::read` in a loop with a delay.

In `main` the `spi0_pins` and `spi0_slave_context` objects are created and ten sets of data obtained from the slave device selected by asserting CE0, specified by passing `spi0_slave::chip0` to the `context` object's constructor. The device requires that each value read is a separate conversation so the call to `spi0_pins::start_conversing` is inside the loop. The only other value explicitly specified for the slave context is the frequency of the clock, given in terms of the frequency types initially created for the clock

Listing 1

```
#include "spi0_pins.h"
#include <array>
#include <thread>
#include <chrono>
#include <iostream>
#include <iomanip>

using namespace dibase::rpi::peripherals;
unsigned char read_byte(spi0_pins & spi0)
{
  constexpr auto a_short_while
    (std::chrono::microseconds{100});
  unsigned char byte{0U};
  while (!spi0.read(byte))
    std::this_thread::sleep_for(a_short_while);
  return byte;
}
bool read(spi0_pins & spi0,std::array<int,2> &
result)
{
  constexpr unsigned char mode{0xd0};
  if (spi0.write(mode) && spi0.write(mode))
  {
    result[0] = read_byte(spi0);
    result[1] = read_byte(spi0);
    return true;
  }
  return false;
}

int main()
{
  try
  {
    constexpr auto f_sclk(megahertz{1});
    spi0_slave_context chip0_context{
      spi0_slave::chip0, f_sclk};
    spi0_pins spi0{rpi_p1_spi0_full_pin_set};
    for (unsigned i=0; i<10; ++i)
    {
      spi0.start_conversing(chip0_context);
      std::array<int,2> data;
      if (read(spi0,data))
        std::cout << std::hex
                  << std::setfill('0')
                  << std::setw(2) << data[0]
                  << ' '          << std::setw(2)
                  << data[1] << '\n';
      else
        std::cout << "## ##\n";
    }
  }
  catch (std::exception & e)
  {
    std::cerr << "Failed because: "
              << e.what() << '\n';
  }
}
```

peripherals' library support described in part 2. To ensure that resources are released should an exception be thrown the whole lot is wrapped in a try-block. The single catch clause for `std::exception` by reference suffices as the library throws standard library exception types or types derived from them.

SPI standard mode reading and writing is tested using a loop-back configuration connecting the MOSI pin to the MISO pin so each written byte is immediately received back again. The support for reading and writing using the 2-wire modes can at best be termed 'provisional' as I have not been able to test them. I did not see how I could use a loopback setup with these modes and had no devices to hand that supported them – nor had I come across any in my very limited search for devices.

I²C – a serial interface by many other names

Having implemented SPI support allowing me to use the DAC and ADC chips on the Gertboard the only remaining device to look at was an Atmel AVR ATmega 8-bit microcontroller which houses a variety of useful interfaces and peripherals. ATmega application programs are stored in non-volatile flash memory and are sent via a supported interface with the microcontroller in a programming mode. On the Gertboard the ATmega microcontroller is programmed over SPI, using SPI0 at the Raspberry Pi end. I contemplated using the microcontroller as a peripheral extender but did not want to dedicate SPI0 permanently to the microcontroller. A browse through the relevant ATmega data sheet [9] revealed a two wire serial interface (TWI) compatible with the Inter-Integrated Circuit (IIC or I²C) interface [10] is supported. Thinking this could be used for Raspberry Pi and microcontroller communication I decided to add I²C support to the peripherals library.

The peripheral documentation for the BCM2835 calls its I²C-like serial interface 'Broadcom Serial Controller' (BSC). Three BSC master controller peripherals – BSC0, BSC1 and BSC2 are supported but only BSC0 and BSC1 are available for use via appropriately configured GPIO pins, BSC2 being reserved for use with the HDMI interface.

As you may have inferred I²C/BSC/TWI uses the master/slave model with each BSC controller acting as an I²C master (I²C slave mode is supported by the same peripheral that supports SPI slave mode). I²C only uses two wires – or pins – referred to as serial data (SDA) and serial clock (SCL). Slave devices have an address which is generally in a 7-bit range, but a cunning scheme can allow 10-bit addressing to be used. This scheme is outlined later. All transfers consist of serialised 8-bit bytes with the master sending an initial start byte containing the 7-bit address of the slave device the master wishes to converse with and a single bit indicating whether the master is reading or writing. Standard I²C can communicate at up to 100,000 bits per second (100 Kbps). The BSC controllers support I²C fast-mode allowing speeds of up to 400Kbps. Like SPI0 there are various parameters that can be adjusted to ensure master and slaves can communicate – the serial clock frequency value for example. All of these parameters I found could have useful default values but unlike SPI0 they apply to a controller as a whole and not on a per slave device basis.

BSC masters use a single 16 entry 8-bit wide FIFO that is shared by read and write operations as the I²C bus cannot be doing both simultaneously. Primarily only polled usage is supported by the BSC masters although interrupts can be generated for some interesting conditions. As with other peripherals the library only supports polled usage.

As there are only the two pins...

Once again, support for the BSC masters broadly follows the pattern discussed in part 2 with the `i2c_registers` class matching the layout of the BSC masters' control registers and allowing querying and setting the various fields. However, the `i2c_ctrl` singleton class deviated from the pattern because the BSC masters' register blocks are located sufficiently distant from each other that an array of three `phymem_ptr<volatile i2c_registers>` was required to map three `i2c_registers` instances over three distinct BSC master register address blocks. As with SPI0, because I²C requires more than one pin there is the `i2c_pins` (plural) class.

I thought two pins were few enough that they could be passed directly to `i2c_pins` constructors as individual parameters. Annoyingly there is a case where the same two pins support two BSC masters on different alternate pin functions. Although the target Raspberry Pi models do not provide access to these pins I wanted to support this case so provided two constructors. One identifies a BSC master from just two passed `pin_id` values while the other is additionally passed a disambiguating 0 or 1 value to indicate BSC0 or BSC1 directly. The remaining parameters, common to both constructors, define a bunch of communication parameters and all have defaults.

Conversation starter

As I²C communications parameters apply to the whole peripheral the conversation state complexity of SPI0 does not apply. The only thing required to talk to a slave device is its 7-bit address, which is sent by the master at the start of a transaction along with a read/write bit. To cater for these transaction start requirements I added `start_write` and `start_read` member functions in addition to write and read member functions.

BSC masters require the data (byte) length – in the range [0, 65535] – to be specified at the start of each transaction. The transaction completes when data-length bytes have been transferred. So the BSC master peripherals require a slave device address and a transaction data length to start a transaction. When starting a write transaction it is useful to pass an initial chunk of data to transfer, however immediately after starting a read transaction there is nothing yet to receive.

Only multi-byte transfers are supported by the read and write operations of `i2c_pins` which return a `std::size_t` indicating how many bytes were transferred. A transfer may not complete because either the FIFO is full so no more data can be written to it or it is empty and no more data can be read from it. The pattern is to call `start_write` or `start_read` followed by repeated calls to `write` or `read` until the transaction completes, preferably with a delay between calls to allow time for data to transfer.

The BSC master peripherals support a variety of status information. The transfer active condition indicates when a transfer is in process and is presented by the `i2c_pins::is_busy` member function which returns `true` while data transfer is ongoing. For finer grained control there are various FIFO states that can be queried.

There are a couple of potential communication errors. A slave device may fail to acknowledge an address and the SCL clock line may timeout. Slaves can stretch clock ticks on SCL within limits – set as one of those communications parameters passed to `i2c_pins` constructors. A slave does this if it cannot respond quickly enough and needs to slow the master's outpourings. The error states can be queried with the `no_acknowledge` and `clock_timeout` member functions. Plagiarising the C++ standard library IOStreams states I also provided a `good` state query member function along with two state `clear` member functions – one clearing both error states, the other clearing specific error states.

Listing 2 shows an example of using `i2c_pins` to write data to a memory device [11], read the values back and display the written and read values and whether they differ. The memory device's write operation starts with the initial address to write to. The device has a 512 byte capacity, a 9-bit range, but the initial address is only 8-bits. The device uses two 256 byte pages to access the whole 512 bytes with page 0 having an even slave address and page 1 the following odd address. The first byte of the transferred data is written to the specified address with following bytes written to subsequent addresses. As the device read operation used does not specify a start address it is handy that addresses wrap round to zero after reaching 511. Reads start at the current address and then from each following address. So writing 512 bytes will return the current address to the initial location – address zero in this case, which if followed by a read will read the previously written data starting with the first byte written.

Listing 2

```
#include "i2c_pins.h"
#include <array>
#include <thread>
#include <chrono>
#include <iostream>
#include <iomanip>

using namespace dibase::rpi::peripherals;

constexpr int wrap_length{512}; // bytes
constexpr unsigned char memdev_addrs{0x50};
  // page 0
typedef std::array<unsigned char,wrap_length>
  buffer_t;
void quick_doze()
{
  std::this_thread::sleep_for
  (std::chrono::microseconds{100});
}
void write(i2c_pins & bsc,unsigned char dev_addrs
          , unsigned char start_addrs
          , buffer_t & data)
{
  bsc.start_write(dev_addrs,data.size()+1
                , &start_addrs,1);
  unsigned char * write_ptr{&data[0]};
  std::size_t remaining{data.size()};
  while (remaining)
  {
    if (bsc.write_fifo_has_space())
    {
      std::size_t transferred
        = bsc.write(write_ptr, remaining);
      remaining -= transferred;
      write_ptr += transferred;
    }
   else
    quick_doze();
  }
  while (bsc.is_busy())
    quick_doze();
}
void read(i2c_pins & bsc
        ,unsigned char dev_addrs,buffer_t & data)
{
  bsc.start_read(dev_addrs,data.size());
  unsigned char * read_ptr{&data[0]};
  std::size_t remaining{data.size()};
  while (remaining)
  {
    if (bsc.read_fifo_has_data())
    {
      std::size_t transferred
        = bsc.read(read_ptr, remaining);
      remaining -= transferred;
      read_ptr += transferred;
    }
    else
      quick_doze();
  }
  while (bsc.is_busy())
    quick_doze();
}
```

In **main** an **i2c_pins** instance is created using GPIO pins 2 and 3 for SDA and SCL respectively (see **pin_id** and friends [12, 1]). This translates to using BSC1 – hence the object's name.

Using direct GPIO pin numbers in this case means knowing the Raspberry Pi revision as the original model B presents GPIO pins 0 and 1 on their P1

Listing 2 (cont'd)

```
int main()
{
  try
  {
    i2c_pins bsc1{pin_id{2},pin_id{3}};
    buffer_t wb;
    for (int v=0; v!=wrap_length; ++v)
      wb[v] = v;
    write(bsc1,memdev_addrs, 0,wb);
    buffer_t rb = {{}};
    read(bsc1,memdev_addrs, rb);
    std::cout << std::boolalpha
              << std::setfill('0')
              << std::hex
              << "Wrote Read Same?\n";
    for (int v=0; v!=wrap_length; ++v)
      std::cout <<   "  " << std::setw(2)
                << int(wb[v])
                << "    " << std::setw(2)
                << int(rb[v])
                <<  "   " << (rb[v]==wb[v])
                << '\n';
  }
  catch (std::exception & e)
  {
    std::cerr << "Failed because: "
              << e.what() << '\n';
  }
}
```

connector's pins 3 and 5 – which support BSC0, while later model B revisions and subsequent models (excepting the compute module) connect GPIO pins 2 and 3 to this pair of pins, supporting BSC1. The problem could be solved by specifying **p1_pin(3)** and **p1_pin(5)** or use the pre-defined objects **sda** and **scl**.

A **std::array** type, under the alias **buffer_t**, is used as the type for buffers. The write buffer is created and filled with values matching the index of each byte. The program's **write** function performs the write operation. It is passed the **bsc1 i2c_pins** object by reference along with the memory device's page 0 address, the memory address to start writing to and the write buffer by reference. The **write** function initiates a write transaction by calling **i2c_pins::write_start** on the passed **i2c_pins** object – specifying the passed-in device address, one more than the write buffer length as the number of bytes to transfer to account for the initial memory page start address and passes the address of the memory page start address argument object as the single byte 'buffer' to initially transfer.

The following loop writes values from the write buffer to the FIFO. If there is space in the FIFO, determined by a call to **i2c_pins::write_fifo_has_space**, data is written to it from the write buffer otherwise the thread waits for space to become available by taking a short sleep – as implemented by the **quick_doze** function. The loop terminates when all the data has been written to the FIFO but the transaction only completes shortly after the FIFO empties. We could call **i2c_pins::write_fifo_is_empty** but that 'completes shortly after' can cause the peripheral to still be busy when the next transaction is attempted so it is best to wait for **i2c_pins::is_busy** to return **false**.

The values are then retrieved by calling the program's **read** function which takes the same parameters as **write** except there is no starting memory address value as the read operation used starts reading from the current memory address. A separate read buffer is passed to read so that both written and read values are available for comparison. The workings mirror those of **write**. The transaction is initiated by calling **i2c_pins::start_read** passing the memory device's address and the size of the passed data buffer to read into. Data is read into the buffer in chunks as they appear in the FIFO. Calls to **i2c_pins::read_fifo_has_data** check there is data to collect

otherwise the thread takes a short doze. Finally **read** waits for **i2c_pins::is_busy** to return **false** indicating transaction completion.

The final action of **main** is to write out the bytes in the write and read buffers and whether each pair of values match.

You might be wondering why, as BSC masters do not have separate read and write FIFOs, why the FIFO checking member functions specify read and write in their names. The same conditions apply to **spi0_pins**, and SPI0 does have separate read and write FIFOs. Thinking some consistency might be nice I re-used the names.

I've not finished so I'll start

The BSC masters support the I^2C repeated start feature allowing a master and slave to conduct multiple transactions without asserting the stop condition and releasing the bus. Repeated start is only practically relevant to read operations where the master has to send some information to the slave device first – a value identifying a device register to be read for example. In such cases the master sends the information on what is requested then enters a start condition, without first going through a stop condition, specifying the same slave address but changing the read/write bit to read.

To support repeated start read operations I added an overload of **i2c_pins::read** that takes the data to write as an additional **std::uint8_t** parameter. Repeated starts are normal (read) transactions issued before the preceding (write) transaction has completed. The documentation is not very clear but it seems repeated starts must be issued after the last byte has started transfer but not completed which is difficult if not impossible to detect for multi-byte transactions. So the initially written data is restricted to a single byte allowing the start of the last and only byte's transfer to be detected by waiting briefly and busily for **i2c_pins::is_busy** to return **true**. When it does the repeated start can be initiated – but has to be setup in the short time before the write transfer completes. The **i2c_pins::read** overload only returns after the byte-write has completed, counter-intuitively by waiting for **i2c_pins::read_fifo_has_data** to return **false**, otherwise the byte transfer may be incompletely and falsely reported as read data.

Because the code runs on a pre-emptively scheduled system there is a chance the processor wanders off to do something else while waiting for the single byte write transaction to start and the whole transaction could have completed by the time the thread is scheduled to run again. This means the window in which **i2c_pins::is_busy** returns **true** is missed and the wait loop will never exit. To prevent this a maximum number of iterations is defined and an iteration count kept. If the count exceeds the maximum, **false** is returned immediately indicating a missed repeated start and the caller should retry. Other error conditions are signalled by throwing an exception.

Listing 3 shows a **read** function that uses repeated starts with a random read operation of the memory device. It differs from the **read** function in listing 2 by having an additional **mem_addrs** parameter which is passed to the repeated-start supporting **i2c_pins::start_read** member function in a loop which permits a number of retries to account for the possibility of failure due to context-switches as discussed above.

The ten-bit cunning scheme

One use of repeated starts is to support I^2C 10-bit addressing. The additional address bits are provided by a byte written before the rest of the transaction, which for write transactions just adds an additional byte to the transaction similar to the listing 2 **write** function. Read transactions from 10-bit addressed slave devices require the extra address byte to be written followed by a repeated read-transaction start as per the listing 3 **read** function. You would think the extra 8-bits would give a 15 bit address but the upper 5 bits of the usual 7-bit part of the address use a fixed pattern specifying a reserved range of addresses, leaving only the lower 2 bits to be available for use as the most significant bits of the extended 10 bit address.

Listing 3

```
bool read(i2c_pins & bsc,std::uint8_t dev_addrs
          ,std::uint8_t mem_addrs, buffer_t &
data)
{
  int remaining_tries{3};
  bool started{false};
  while (!started)
  {
    started = bsc.start_read(dev_addrs
                             ,mem_addrs,data.size());
    if (!started && --remaining_tries==0)
      return false;
  }
  std::uint8_t * read_ptr{&data[0]};
  std::uint32_t remaining{data.size()};
  while (remaining)
  {
    if (bsc.read_fifo_has_data())
      {
        std::uint32_t transferred
          = bsc.read(read_ptr, remaining);
        remaining -= transferred;
        read_ptr += transferred;
      }
     else
      quick_doze();
  }
  while (bsc.is_busy())
    quick_doze();
  return true;
}
```

Wrapping up

So that's about it. I think C++, and C++11, features proved useful in providing simple interfaces to GPIO and other peripherals. The resultant overall structure of the library appears to allow easily adding support for other peripherals or even other modes of peripherals having existing support – such as the serialiser mode of the PWM controller.

There are some parts that have not worked out quite as well as they could. The pin and peripheral allocation support is I think the part with the most problems. While tracking which pins and peripherals are in use is a good thing in theory as there is no system wide way of achieving this the partial solutions currently implemented by the library leave quite a lot to be desired.

While developing the initial implementation I intentionally ignored concurrency and synchronisation issues. While the library can with care be used in a multithreaded environment there should be a review of concurrency concerns at the very least – especially as the latest Raspberry Pi 2 model B has a 4 core chip.

Which leads on to supporting the growing list of Raspberry Pi models. The main concern is that the new BCM2836 based Raspberry Pi 2 model B maps peripheral registers to a different base address. Other concerns are supporting the additional GPIO pins on the larger 40 pin connector and detecting which model code is running on. I have added some preliminary support for these things to the **pin_id** family of classes and for use by the **rpi_info** type to determine the board revision details.

But before rushing into too much new functionality it is probably a good point to review the interfaces provided for each peripheral as well as the code in general and of course those 2-wire modes of **spi0_pins** still need to be tested. ■

References

[1] Raspberry Pi Linux User Mode GPIO in C++ (Part 1), *CVu*, Volume 27 Issue 2, May 2015

[2] Raspberry Pi Linux User Mode GPIO in C++ (Part 2), *CVu*, Volume 27 Issue 4, September 2015

One Definition Rule

Roger Orr explains an often misunderstood aspect of C++.

C++ allows a lot of flexibility over the physical arrangement of source code. Almost all C++ programmers have a concept of 'source file' and 'header file' and some views about what should be placed into each file. What does this mean in practice?

There is no single simple answer in modern C++.

Traditionally people have put function declarations, class definitions, and constants into header files and placed function and member function implementations into source files. However, the use of templates and the presence of inline functions, where the full definitions are normally required whenever they are referenced, has blurred the distinction between headers and source files.

Conversely, many tool chains now offer some sort of 'whole program' optimisation which means that the linker has an overview of all the source code in the program and again this reduces the clear distinction between separate source files since cross-source optimization is now possible.

The 'One Definition Rule' (ODR) is attempting to ensure that there is at most one definition of the various entities (classes, functions, etc.) in each **source** file and also ensure that there is a single unambiguous definition in the resulting whole **program**. While both of these are covered by the ODR the first case is much easier for the compiler (and the programmer) to detect than the other. It's the second case, that of the same entity being defined differently in two different components of the same program, that can cause very hard-to-diagnose problems.

The language standard itself speaks in terms of 'translation units' by which it means a source file, together with all the lines of code in headers and other source files `#included` by it, excluding any lines skipped by any of the conditional inclusion preprocessing symbols. This *textual* inclusion model means that the same lines of text from the same ('header') file can be included in multiple translation units (TUs) but each is compiled separately and the context for compilation will be provided by the previous lines of text, the preprocessing symbols defined, and any additional switches provided to the compiler.

This is a very different model from many languages where inclusion is a reference to the *compiled* output from another source file. In these cases there is no ambiguity about where things are defined nor about what the context was for their compilation. While it is still possible to cause problems, for example by using a different version of the included file during compilation from that used at runtime, the problems are greatly reduced.

The ODR refers to the 'same sequence of tokens' which is not necessarily equivalent to 'from the same (header) file' and there are additional constraints basically trying to ensure that if the same thing is defined in two different translation units the two different compilation contexts have not affected the meaning of the entity being defined.

Demonstration of the problem

Here we can see the same named structure, `Data`, is used in both source files but with a different number of fields in the two cases. The behaviour

Listing 1

```
struct Data
{
   int id;
   std::string first;
   std::string second;
   std::string last;
};
extern Data getData(int employeeId)
{
   // Implementation details omitted...
}
```

Listing 2

```
struct Data
{
   int id;
   std::string first;
   std::string last;
};
extern Data getData(int employeeId);
int main()
{
   int id;
   std::cout << "Enter employee Id: ";
   std::cin >> id;
   Data const details( getData(id) );
   std::cout << "employee " << id << " is "
             << details.first << std::endl;
}
```


ROGER ORR

Roger has been programming for over 20 years, most recently in C++ and Java for various investment banks in Canary Wharf and the City. He joined ACCU in 1999 and the BSI C++ panel in 2002. He may be contacted at rogero@howzatt.demon.co.uk

Raspberry Pi Linux User Mode GPIO in C++ (Part 3) (continued)

[3] dibase-rpi-peripherals library project: https://github.com/ralph-mcardell/dibase-rpi-peripherals

[4] Resource acquisition is initialization (RAII), see for example: http://en.wikipedia.org/wiki/Resource_Acquisition_Is_Initialization

[5] Gertboard Raspberry Pi IO expansion board: http://www.raspberrypi.org/archives/411

[6] Serial peripheral interface (SPI), see for example: https://en.wikipedia.org/wiki/Serial_Peripheral_Interface_Bus

[7] BCM2835 ARM Peripherals: http://www.raspberrypi.org/wp-content/uploads/2012/02/BCM2835-ARM-Peripherals.pdf

[8] Master/slave model, see for example: https://en.wikipedia.org/wiki/Master/slave_(technology)

[9] ATmega48A/PA/88A/PA/168A/PA/328/P 8-bit microcontroller datasheet: http://www.atmel.com/images/Atmel-8271-8-bit-AVR-Microcontroller-ATmega48A-48PA-88A-88PA-168A-168PA-328-328P_datasheet_Complete.pdf

[10] Inter-Integrated Circuit (I^2C): http://www.nxp.com/documents/user_manual/UM10204.pdf

[11] Cypress FM24CL04B 4-Kbit (512 x 8) Serial (I^2C) F-RAM http://www.cypress.com/file/136466/download

[12] The `pin_id` class et al in `pin_id.h`: https://github.com/ralph-mcardell/dibase-rpi-peripherals/blob/master/include/pin_id.h

of the resultant program if source file 1 and 2 are linked together will be **undefined**.

I tried two different compilers and in both cases I got output something like this:

```
Enter employee Id: 1
employee 17370356 is Roger
```

While the actual number printed varied, in neither case did I get any warnings nor any crashes, but the employee Id printed out *wasn't* the value that was supplied.

With a third compiler I got this:

```
Enter employee Id: 1
employee 1 is Roger
```

followed by a crash of the program.

What has occurred in the first two cases is that the call to **`getData` corrupted** the value held in **`id`**. This is because the data structure **`details`** has enough space for two strings, but the function in source file 1 is populating the structure with three strings, and this is overwriting the id value as it just happens to be the next address in memory beyond the end of the **`details`** structure.

In the last case it seems that the corruption is still occurring, but the memory layout in this case is different and the corrupted value is not **`id`** but something else, such as the return address, causing the crash when **`main`** ends.

In this example it is easy to see what the problem is; in a real-world example it can be extremely hard to detect the problem.

How can we prevent this happening?

The problem above was two different definitions of the same data type. While ODR violations affect many different entities in C++ one of the most troublesome is when the layout of a class changes as the consequences of this can be quite unexpected and hard to isolate. ODR violations with other entities still result in undefined behaviour, but in my experience the actual consequences at runtime tend to be easier to understand.

We can reduce the likelihood of the definitions for the data type differing by placing one definition inside a single file and including this header file in both the source files.

That is fairly obvious; but it is hard to enforce. It is quite common to have multiple **copies** of the same header file in the source tree for a large project – perhaps caused by laziness or short-term desire for 'simplicity'. Unfortunately, as we all know, once you have multiple copies they tend to diverge and then you have different definitions in the various files.

One common cause of this problem is when two different versions of a 3rd party library are used in the same project – perhaps the project includes a library that was implemented with one version of Boost and the project code accidentally uses a different version.

It is also common for people to create various helper classes and functions for use in the *implementation* of a class. These entities are defined in the source file itself and are not visible outside it. However, if another source file happens to define the an entity with the same name then we have an ODR violation. While many of these are benign (as the code is localised and often gets completely inlined) the other cases can cause nasty bugs.

Placing helper classes and functions into the anonymous namespace avoids ODR violations like these and should be encouraged. In my experience people are becoming fairly good at putting helper functions into an anonymous namespace but for some reason less commonly place helper classes into a namespace.

However, even with a class defined in a single header file in one location there are still a number of ways that differences can creep in.

In my experience the commonest problems are caused by:

1. preprocessor symbols
2. packing
3. compiler flags

Different preprocessor symbols

It is common to see header files set up to support a number of different compilation modes. For example, some data structures may need locking with a mutex in a multi-threaded program but do not need this protection in a single-threaded one. One common implementation technique is to conditionally define the mutex, and the associated code using it, based on a preprocessor symbol that the user of the header file defines if required.

A sample data structure might look like Listing 3.

The danger here is that if *some* of the source files in the program that include this header file have **`MULTI_THREADED`** defined and *some* do not then we have an ODR violation because of the different number of fields and the resulting program may not behave as we would like.

However, as in the example I started with, the actual consequences of the undefined behaviour are very hard to predict and are likely to vary between compilers.

Different packing

Compilers align fields on a 'natural' address to fit in with the addressing modes of the underlying architecture. So for example, on a 64-bit platform the first couple of fields in the data structure above might be laid out as:

```
int id; // offset 0, length 4
std::string first; // offset 8, length 32
```

However, many compilers allow the programmer to use a pragma or attribute to override this natural alignment and to pack the fields as close together as possible. This typically results in a reduction in memory and may also result in a reduction in performance, depending on a number of factors. Packing is also used when ensuring a C++ data structure matches some externally defined data structure, perhaps from a serialisation format or network protocol.

The way to do this in Microsoft Visual C++ is to use the **`#pragma`** pack directive, for example:

```
#pragma pack(2) // 2 byte aligned
#include "data.h"
#pragma pack() // back to the default
```

with these packing instructions the fields might now be laid out as:

```
int id; // offset 0, length 4
std::string first; // offset 4, length 32
```

However, mostly to provide compatibility with MSVC, other compilers such as gcc also support the same pragma and it is quite common to see the pragma used even in code used cross-platform.

Once again, as long as *every* source file in the program uses the same packing value we are fine, but if just one file does not set the same packing then we cause an ODR violation.

This problem is much less likely to occur with the idiomatic style provided by gcc which uses attributes attached to the *specific* data structure(s) they are applied to.

```
class Data {
   // ...
} __attribute__ ((packed));
```

Listing 3

```
class Data
{
public:
    // Methods ...
private:
   int id;
   std::string first;
   std::string second;
   std::string last;
#ifdef MULTI_THREADED
   std::mutex mutex;
#endif;
};
```

One further trouble with the example shown above using **#pragma pack** is that we don't know what other header files might be included by the line **#include "data.h"**. If, as is quite likely, `data.h` in turn **#include**s **<string>** then by setting the packing while including `data.h` we may have inadvertently caused the standard string class to be packed too.

I would recommended if at all possible that you ensure that **#pragma pack** directives are very carefully scoped to ensure that no header files are **#include**d within the scope of the directive.

Different compiler flags

Depending on the compiler being used there may be various flags which can be set during compilation which change the layout of the class. Often these flags are simply another way of generating a different set of preprocessor symbols, or different packing, but there are some other cases too – for example specifying the size of the **long double** type or whether the **char** type is **signed** or **unsigned**.

It is important to ensure the set of flags used in building a single program are consistent where the use of different settings could result in ODR violations.

This relates to build system discipline where it is advisable to set program-wide settings at the top level of the build system rather than individually specifying them for each component or even for each file.

And more...

An additional problem is when the meaning of **types** change between source files.

For example, a structure is defined with a field **char buffer[BUFSIZE];** but the value of **BUFSIZE** varies between source files.

More subtle problems can occur when, for instance, a type defined in one namespace masks a type defined in an outer namespace so a different type is used for the field in two different files.

Can the compiler help?

If the ODR violation occurs inside a single translation unit then we do expect, rightly, that the compiler will error on ODR violations. The standard is quite clear: "No translation unit shall contain more than one definition of any variable, function, class type, enumeration type, or template."

In the cases of ODR violation between TUs it is much harder to see how the compiler – which compiles each TU independently – can assist us. However, in some cases the use of name mangling can help, for example to avoid linking functions with mismatched calling conventions together.

-Wodr to the rescue

Recent versions of gcc support link-time optimisation (lto) with the **-flto** flag which allows the linker access to the internal bytecode generated by the separate compilations. This allows the linker to perform additional optimisations, such as inlining a function call from one source file to another.

The mechanism has been slightly extended in gcc 5.x to include support for checking for some of the ODR violations covered above for data types; the byte code generated by each source file compiled with lto support will contain additional information about the data structures involved and this can be checked at link time for consistency.

If I compile the example I started with using gcc 5.2 and **-flto**, I get the error in Figure 1at **link** time.

(Note that for **-Wodr** to work, **both** the `.o` files with the differing type definitions must be compiled with **-flto**.)

This is extremely useful and has helped me to locate several places in a large code base where there were previously unidentified ODR violations, at least one of which had led to large memory leaks in the past which had been worked round without actually finding the root cause of the problem.

Modules

As many of the readers may already know, there is active work towards a 'modules' system for C++, which among other things should make it easier to avoid ODR violations, especially with internal 'implementation only' entities used purely inside the library. However, a lot will depend both on the final form of the proposals and also on the details of the implementation strategies adopted by the compiler or compilers you use.

Summary

ODR violations can be very hard to spot. Good programming discipline in:

- naming things
- avoiding copy-paste (see the DRY principle)
- using namespaces
- using anonymous namespaces in implementation files
- build system settings

all help to reduce the likelihood of experiencing ODR violations.

The gcc **-Wodr** detection included in link-time optimisation is highly recommended (at least in gcc 5.2 and above – there did appear to be some occasional problems in 5.1) and it might be worth using this as a static analysis tool even if the primary build doesn't use link time optimisation (or even uses a different compiler). ■

Figure 1

```
$ g++ -Wall -Wextra -flto -o SourceFile1.o SourceFile1.cpp -c
$ g++ -Wall -Wextra -flto -o SourceFile2.o SourceFile2.cpp -c
$ g++ -o Program SourceFile1.o SourceFile2.o
SourceFile1.cpp:2:8: warning: type 'struct Data' violates one definition rule [-Wodr]
 struct Data
        ^
SourceFile2.cpp:4:8: note: a different type is defined in another translation unit
 struct Data
        ^
SourceFile1.cpp:5:17: note: the first difference of corresponding definitions is field 'second'
     std::string second;
                 ^
SourceFile2.cpp:7:16: note: a field with different name is defined in another translation unit
    std::string last;
                ^
```

Functional Programming in C++

Richard Falconer reports on an ACCU talk by Kevlin Henney.

A common belief is that functional programming is constrained to the realm of Haskell and Lisp. On 14th September, Oxford Asset Management hosted a talk by Kevlin Henney showing that this is not the case, and that you can in fact follow functional programming paradigms in C++.

If we had a time machine the first thing any of us would do is go back in time and make **const** the default modifier. Well, maybe not the first thing. After all one must spare a thought to how the C++11 committee would deal with re-purposing the original **auto** as a result of such temporal meddling. The point, Kevlin explains, is pure functional programming has no side-effects.

When it is not necessary to change, it is necessary not to change
~ Lucius Cary

To move towards functional programming is to move towards a stateless world, a world where moving parts of a system should be isolated and data flows in one direction through classes. 'Values' are a core concept of functional programming; rather than objects with a stateful identity these are objects with unchanging values. Kevlin gives the canonical example of a poorly-written **Date** class (see Listing 1, based on slides 37 and 39).

Here it's clear that by calling the set methods you can leave the object in an invalid state; 30th Feb never occurs unless you're PHP (which thinks Feb 30th is basically the same as March 2nd [1]).

Instead, Kevlin points out, you should rebind state via the assignment operator rather than permuting the existing object (see Listing 2, based on slide 40).

Now this is a value-semantic class whose instances can be passed around and referenced without worrying about race conditions or other threads modifying our object through C++'s many aliasing mechanisms.

This isn't to say that we should do away with all state, but minimising it allows you to reason about the interaction between functions more easily; there's no need to keep your mental buffer busy tracking side-effects and then lose 20 minutes work when someone comes to your desk to ask you if you received the email they just sent.

Now that we have stateless classes (or at least, classes whose state does not change from the perception of the class API boundary) we can start to build by composition. A core requirement for functional programming is to have functions as first-class citizens; that is, functions as things you can return and accept as arguments. This function-passing allows complex behaviour to be *composed* by expressing ideas in terms of combinations of other functions. This composition is made syntactically easier through the use of lambdas, which are now available in C++11. (C# has boasted true lambda support since v3.0 2007, but everyone is too polite to mention they've been around as a concept since Alonzo Church's 1932 journal [2] in *Annals of Mathematics*).

Listing 1

```
class date
{
public:
  date(int year, int month, int day);
  int get_year() const;
  int get_month() const;
  int get_day() const;
  int set_year(int);
  int set_month(int);
  int set_day(int);
void set(int year, int month, int day);
private:
  ...
};
auto today = date(2015, 9, 15);
today.set_day(16);
```

Listing 2

```
class date
{
public:
  date(int year, int month, int day);
  int year() const;
  int month() const;
  int day() const;
};
auto today = date(2015, 9, 15);
today = date(2015, 9, 16);
```

Evaluation of our functions objects does not depend on mutable state. Expressions created by composing such objects are said to be referentially-transparent; the expression could be replaced by its value without changing the behaviour of the program [3].

Asking a question should not change the answer
(Nobody tell Heisenberg.)

This is all very well for simple value types like **Date**, but what about something less trivial?

Persistent Data Structures [4] (not to be confused with a persistent storage) are effectively immutable (see Listing 3, based on slide 56).

Note **popped_front**/**back** are const. They return the view of the data with those operations applied: "what would you look like with the front popped". The underlying data is unpermuted, and any aliases to the original full vector can still resolve. You can then have many views on the

Listing 3

```
template<typename T>
class vector
{
public:
  typedef const T * iterator;
  ...
  bool empty() const;
  std::size_t size() const;
  iterator begin() const;
  iterator end() const;
  const T & operator[](std::size_t) const;
  const T & front() const;
  const T & back() const;
  const T * data() const;
  vector popped_front() const;
  vector popped_back() const;
private:
  ...
  iterator from, until;
};
```

RICHARD FALCONER

Richard Falconer is a C#/C++ developer with interests in Security, UI design, and Juggling. He can be contacted at richard@rjfalconer.com or @rjfalconer.

same data, with no need to copy the full vector. (The vector returned from `popped_front` is just the original vector with the internal iterator modified to point to the 2nd element).

This all greatly simplifies the addition of threads. When people think 'Threads', they think 'Locks', yet locks just make your thread wait, and all computers wait at the same speed. Even `shared_ptr` introduces interlocked increments/decrements for all its reference counting operations, but letting threads lose on a code base *without* sufficient const-correctness is tantamount to releasing a pack of rabid dogs through your code. Or has results similar to the times when you introduce a `void*` and tell the compiler "hold my beer and watch this".

> Some people, when confronted with a problem, think "I know, I'll use threads", and then havthey e two prbolems.

With our immutable value-types and generous use of `const` this is all a lot easier.

The solution is to compose from scratch without thinking about locks, and thus be in a situation where your data is immutable or unshared, or both. Kevlin recalls a consultancy job where introduction of one-way data-flow and proper composition reduced the total number of locks used by the system from 30,000 to 6.

Kevlin closes with some thoughts on memory management, which was a problem apparently plaguing England even in the time of Shakespeare's plays:

> Hamlet: From the table of my memory I'll wipe away all trivial fond records.

Clearly Hamlet is a garbage-collection fan.

> Ophelia: 'Tis in my memory locked, and you yourself shall keep the key of it.

Whereas Ophelia prefers reference counting.

Memory management is especially significant here because of the different consumers sharing the same state. Consider a referentially-transparent list class such as Listing 4 (from slide 63).

This is much the same concept as the vector example but with all the operations on front instead. Multiple lists are built up to form an inverted tree structure, with the root node in the tree being the last element of all lists (see Figure 1).

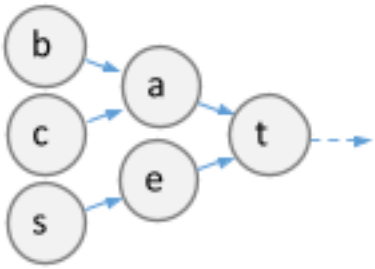


Figure 1

Kevlin demonstrated this with some entertaining audience participation, with Nigel doing a good job of representing the dangling pointer.

The problems start when a list's destructor runs on the leaf node of a long chain. If the reference count of all nodes in that chain are just 1 they too must run their own destructors:

```
{
  list<anything> chain;
  std::fill_n(std::front_inserter(chain),
     how_many, something);
} // What happens when we reach this brace?
```

So the destructors now run recursively, blowing up the stack very quickly (even with small lists in the order of thousands of elements). As a result we have to fallback to garbage-collection when working with these referentially-transparent structures. Notably the Standard allows for garbage collection, but that is a subject for another day.

Listing 4

```
template<typename T>
class list
{
public:
  class iterator;
...
std::size_t size() const;
  iterator begin() const;
  iterator end() const;
  const T & front() const;
  list popped_front() const;
  list pushed_front() const;
private:
  struct link
  {
    link(const T & value,
         std::shared_ptr<link> next);
    T value;
    std::shared_ptr<link> next;
  };
  std::shared_ptr<link> head;
  std::size_t length;
};
```

Our thanks goes out to Oxford Asset Management (especially Charlie, Charlotte, and Tom) for hosting us at their great venue and for providing the buffet. Around 50 people attended; a new ACCU Oxford record.

More information

Kevlin's slides: http://www.slideshare.net/Kevlin/functional-c

ACCU Oxford Meetup page: http://www.meetup.com/ACCU-Oxford/events/223715720/

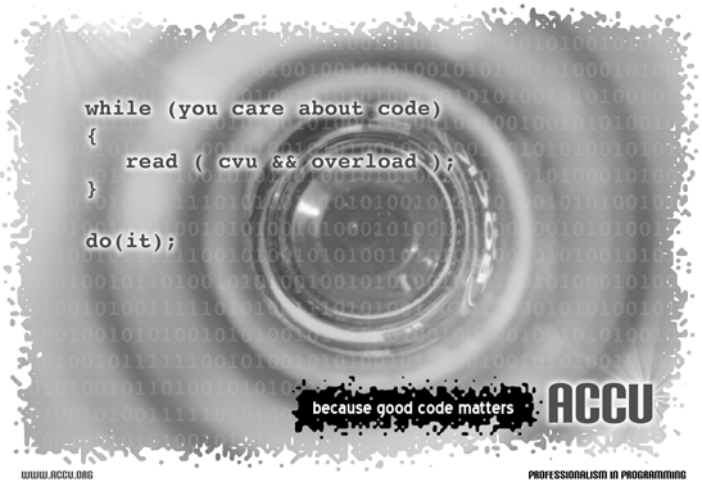
References

[1] `<?php echo date("M-d-Y", strtotime('2015-02-30'));`, although `checkdate(2015,02,30)` does at least return `false`.

[2] A. Church, 'A set of postulates for the foundation of logic', *Annals of Mathematics*, Series 2, 33:346–366 (1932).

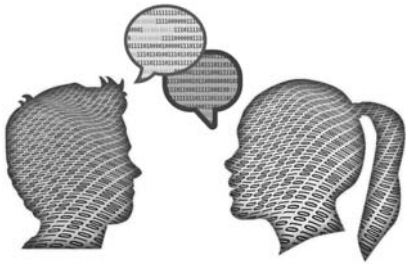
[3] Referential transparency – https://en.wikipedia.org/wiki/Referential_transparency_%28computer_science%29

[4] Persistent Data Structures – https://en.wikipedia.org/wiki/Persistent_data_structure



Code Critique Competition 96

Set and collated by Roger Orr. A book prize is awarded for the best entry.



Participation in this competition is open to all members, whether novice or expert. Readers are also encouraged to comment on published entries, and to supply their own possible code samples for the competition (in any common programming language) to scc@accu.org.

Note: If you would rather not have your critique visible online, please inform me. (We will remove email addresses!)

Last issue's code

I have some C code that tries to build up a character array using **printf** calls but I'm not getting the output I expect. I've extracted a simpler program from my real program to show the problem.

With one compiler I get **"Rog"** and with another I get **"lburp@"**.

I'm expecting to see:

```
"Roger:  10
 Bill:   5
 Wilbur: 12"
```

What have I done wrong?

Can you give some advice to help this programmer?

The code is in Listing 1.

Listing 1

```
#include <stdio.h>
#define ARRAY_SZ(x) sizeof(x)/sizeof(x[0])
typedef struct _Score
{
  char *name;
  int score;
} Score;

void to_string(Score *scores, size_t n,
  char *buffer, size_t len)
{
  for (size_t i = 0; i < n; i++)
  {
    size_t printed = snprintf(buffer, len,
      "%s:\t%u\n",
      scores[i].name, scores[i].score);
    buffer += printed;
    len -= printed;
  }
}
void process(char buffer[])
{
  Score sc[] = {
    { "Roger", 10 },
    { "Bill", 5 },
    { "Wilbur", 12 },
  };
  to_string(sc, ARRAY_SZ(sc),
    buffer, ARRAY_SZ(buffer));
}
int main()
{
  char buffer[100];
  process(buffer);
  printf(buffer);
}
```

Critiques

Mathias Gaunard < mathias@gaunard.com>

The main problem of this snippet is the **ARRAY_SZ** macro, meant to compute the size of an array. This macro will accept pointers as input but provide the wrong answer, in this case **sizeof(char*)/sizeof(char)**, which is the word size, 4 bytes for 32-bit systems. This explains why the result is **"Rog"** on some systems; only 4 bytes were written, 3 characters plus the null byte.

With C++, it is possible to write a function that provides the same functionality but that will lead to errors whenever pointers are passed, by passing the array by reference and using templates to deduce its size:

```
template<class T, size_t N>
size_t array_sz(T(&)[N]) { return N; }
```

By using this function instead of **ARRAY_SZ**, we can easily isolate the errors. One way to fix this is to modify **process** to also take the array by reference.

```
template<size_t N>
void process(char (&buffer)[N]);
```

This will make the code work and display the expected result, however, it still has an issue with how it handles the case where the buffer is not large enough to hold all score listings. As we saw earlier, the code incorrectly claimed that the buffer was word-size-sized, but it should still have always given **"Rog"** as a result, and never **"lburp@"**. **snprintf** can return a negative value on failure, and returns the number of characters it would have written if the buffer is not big enough. It is easy to trigger erroneous output like **"lbur"** simply by hardcoding **printed** to **(size_t)-1**.

It is therefore necessary to do

```
printed = min(printed, len)
```

to avoid writing past the end or before the start of the buffer. Alternatively, one could use dynamically-sized buffers to not impose arbitrary limits, which can be done easily by using C++ iostreams.

Other miscellaneous issues:

- **_Score** is a name reserved for the implementation, and should not be used. All names starting with an underscore followed by an uppercase letters are reserved. By using C++, the typedef/struct becomes redundant anyway, so one should just use **struct Score**.
- The type of the **name** member in **Score** should be **const char*** rather than **char***, since it is initialized from string literals, which are **const** in C++, and shouldn't be modified regardless since it is undefined behaviour to do so.
- The first argument to **printf** should be a format, it is therefore potentially dangerous to call **printf(buffer)** directly in case we ever chose to put special characters in the score entries. Instead, one should write **printf("%s", buffer);**.

ROGER ORR

Roger has been programming for over 20 years, most recently in C++ and Java for various investment banks in Canary Wharf and the City. He joined ACCU in 1999 and the BSI C++ panel in 2002. He may be contacted at rogero@howzatt.demon.co.uk

Peter Sommerlad <psommerl@hsr.ch>

First of all, if a C program misbehaves, I suggest compiling it with a C++ compiler.

Sidenote: Well, that is what I would do and also I would highly recommend to no longer use C at all. With the exception of very, very small targets in the embedded area (8 bit controllers), decent C++ implementations should be available and C should no longer be taught to students. It just lacks any means of abstraction and is too hard to use correctly and even if it is, it is too easy to break things by 'maintenance'.

Now to the problem at hand. Our IDE Cevelop and my C++ compiler tells me several problematic points, including the culprit of the underlying issue.

Let us start our journey in **`main()`**:

```
int main()
{
  char buffer[100];
```

The array line is something I wouldn't have done in C++ at all and I was recommending of not allocating arrays on the stack in C at all, if you can not control the size written. Code like that is behind most security vulnerabilities in the past decades. Cevelop (set to parse as C++14) will claim

- Found C-Array: {0}
- Un- or ill-initialized variable found

The first note is that we try to rewrite code using plain arrays to use the better and copyable alternative **`std::array`** instead (or **`std::string`**). The second comes from our C++11 checker that looks for variables not initialized with the initializer list syntax and transforms them. Even in C code this might be a good idea, since the memory of buffer can contain arbitrary values (only by luck some bytes are 0 if we use C-style string handling in the general case).

Before we apply the quick fixes let us go further to the code of **`main()`**.

```
  process(buffer);
```

Here an array is passed as a function argument and our student should know that any array passed as such degenerates to a pointer to the first element and the dimension of the array is lost. This is still true in C++ and therefore I recommend to never use a plain array as a function parameter or as an argument. It just works without any safeguards and having a programmer deal with array bounds checking explicitly is very error prone and again leads to another big share of software vulnerabilities.

```
  printf(buffer);
```

Even the innocent looking last line is problematic. The **`printf`** function takes at least one **`char`** pointer argument, but interprets its content. Since we will not know what content our process function is storing in the buffer array, we do not know if **`printf`** would be expecting more arguments, because buffer contains any special **`printf`** formatting symbols, such as **`"%s"`**, for example. This could lead to leaking unintended information or in the better case to program crashes, because of illegal memory addresses accessed. It is a similar problem as with the XKCD comic (https://xkcd.com/327/) with **`printf`** syntax instead of SQL.

A better C function to output a plain C string would be

```
  puts(buffer);
```

instead, which at least just expects buffer to be **`NUL`** terminated.

Wow, a full screen full of text with just 3 lines of code. Roger selected a good example this time!

Now let us look one step down in the call chain to process:

```
  void process(char buffer[])
```

as said above, this array parameter is equivalent to writing it as a pointer parameter

```
  void process(char *buffer)
```

and that will make the real error below (stay tuned) much more evident.

```
{
  Score sc[] = {
    { "Roger", 10 },
    { "Bill", 5 },
    { "Wilbur", 12 },
  };
```

The lines above again triggers two messages, one from our plug-in and one from the C++ compiler:

- Found C-Array: {0}
- ISO C++ forbids converting a string constant to '**`char*`**'

The first one tells us again that plain arrays should be verboten. We can automatically change that to use std::array<Score,3> instead, but that would be C++ already. The second warning tells us something about the differences between C and C++, where C++ makes string literal const char arrays, whereas classic C made those char pointers (even without const, which wasn't invented then). So in theory, one could overwrite the literal values for the names given here. This also tells something about the type Score that keeps plain char pointers and no size information. In production code, this is a no-no, because you can not make any safe memory management around such a string representation.

And now to the definitive culprit.

```
    to_string(sc, ARRAY_SZ(sc),
     buffer, ARRAY_SZ(buffer));
  }
```

`ARRAY_SZ(buffer)` expands to

```
  sizeof(buffer)/sizeof(buffer[0])
```

and this generates the compiler warning:

> 'sizeof' on array function parameter 'buffer' will return size of 'char*' [-Wsizeof-array-argument]

which we can compute on a 64 bit computer to become 8 instead of the 100 our student expected and the equivalent of the array with the pointer makes it obvious. **`to_string`** is by the way a bad name for new code, because it is now a function overload in C++'s **`std`** namespace and could make problems, when this code is naively ported to C++ with a using namespace **`std`** in scope (BTW, Cevelop provides a refactoring away from such practice). Cevelop also suggest to change the macro for **`ARRAY`** to a C++ inline (template) function and can do so. In addition this will avoid the potential bug, since the macro is not having parenthesis around its expansion and not around the second use of the parameter. This can cause parsing havoc if used with non-literals as arguments.

So a more correct version of

```
  #define ARRAY_SZ(x) sizeof(x)/sizeof(x[0])
```

would be

```
  #define ARRAY_SZ(x) (sizeof(x)/sizeof((x)[0]))
```

but even better in C++11:

```
  template<typename T1>
  inline constexpr auto ARRAY_SZ(T1&& x) ->
    decltype(sizeof (x) / sizeof (x[0]))
  {
    return (sizeof (x) / sizeof (x[0]));
  }
```

But again that is to no avail in the correct working of process. There are several problems to be resolved with its design. First, passing the buffer from the outside, requires cautions use of its memory resource, so we need to pass the size of the available space. But there is still a problem, we do not know, if the size was sufficient or not. In such a case, it might be better to return an indication if the size was sufficient. This could be a **`bool`**, denoting success (not very helpful), the size actually used and an error code, if insufficient (i.e., -1) or best, the size required for the output, if it was too small, which could be a burden to implement and to use. In any case, passing in memory (buffer) to be used as function output is a hassle. In C++ I would recommend to use an **`ostringstream`** which will

automatically expand an underlying string object and return that by value. This is one area where C is really a burden vs. C++.

It is also unrealistic, that the **Score** array **sc** is local to the function, so this must be passed as well, again explicitly passing its size, meaning we need define 4 parameters and pass 4 interdependent arguments. Using a **std::vector<Score>** in C++ would simplify that again.

Now to the last function in the code where the student actually shows, that she/he understands a bit about passing arrays as pointers with an explicit size. But again, such an API (inherent in C) is easy to use wrongly (as we have seen) by passing inconsistent sizes. Also a return type of **void** signals side effects, the code can have quite unintended ones.

```
void to_string(Score *scores, size_t n,
  char *buffer, size_t len)
{
  for (size_t i = 0; i < n; i++)
  {
    size_t printed = snprintf(buffer, len,
      "%s:\t%u\n",
      scores[i].name, scores[i].score);
    buffer += printed;
    len -= printed;
  }
}
```

Again, here C gets in the way of safe usage of memory. First **snprintf**'s return type is **int**, and there are obscure cases, where it might return a negative number. Second, if the space allowed is too small, it still returns the number of characters that would be needed, so the code will miscalculate the value of **len**, which might underflow. While underflow and overflow does not trigger undefined behaviour with unsigned types, such as **size_t**, the resulting value is still well beyond the capacity of the buffer and thus can easily lead to overwriting memory far away from the allotted memory, resulting in another severe security issue or potential crashes from overwriting return addresses.

A quick fix, would be to at least check if printed is smaller than **len** and only then proceed and otherwise return from the function.

```
if(printed >= len) return;
```

I think I covered most of the problems by now. So how would a refactored C++ version look?

Here is one, that uses the recommendation I put above and getting rid of most of the bad C habits. Its overall design is still not what I consider splendid, but without more context it would be hard to judge if **Score** should have a constructor, be **const** or have member functions/related functions, such as an overloaded output **operator<<**.

```
#include <cstdio>
#include <sstream>
#include <vector>
struct Score
{
  std::string name;
  int score;
};
using ScoreList=std::vector<Score>;
std::string to_string(ScoreList const &scores)
{
  std::ostringstream out{};
  for (auto const &score:scores)
  {
    out << score.name << '\t'
        << score.score << '\n';
  }
  return out.str();
}
std::string process()
{
  ScoreList sc {
    { "Roger", 10 },
    { "Bill", 5 },
    { "Wilbur", 12 },
  };
  return to_string(sc);
}
int main()
{
  std::puts(process().c_str());
}
```

Hope that helps some programmers to become better programmers and write less code.

Gareth Ansell <gareth.ansell@sky.com>

While trying to solve this puzzle I was initially tempted to dive straight for the compiler and start hacking away at the code. However, I managed to resist this urge and engaged brain instead. After which the solution was quite easy to spot. I then modified the code to test my hypothesis, which was proved correct.

The initial problem is in the **process()** function, where the buffer array is passed as an argument. Since this is decomposed to a pointer, it is not possible for the subsequent call to the **to_string()** function to determine the size of **buffer[]** by using **sizeof** (in the **ARRAY_SZ** macro).

In C the solution to this is to add a **length** parameter to the **process()** function, and use this in the call to **to_string()**. In C++, a templated solution could be used.

Apart from this there are a few minor niggles:

1. In **_Score score** is an **int**, but in the call to **snprintf** it is referenced as an **unsigned**.
2. The last element of the **sc[]** array in **process[]** has a trailing comma
3. The **for** loop in **to_string()** compares a signed to unsigned **int**.

My working solution is shown below:

```
#include <stdio.h>
#define ARRAY_SZ(x) sizeof(x)/sizeof(x[0])
typedef struct _Score
{
  char *name;
  unsigned int score;
} Score;
void to_string(Score *scores, size_t n,
  char *buffer, size_t len)
  {
    for(unsigned int i = 0; i < n; i++)
    {
      size_t printed = snprintf(buffer, len,
        "%s:\t%u\n",
        scores[i].name, scores[i].score);
      buffer += printed;
      len -= printed;
    }
  }
void process(char buffer[], size_t len)
{
  Score sc[] = {
    { "Roger", 10 },
    { "Bill", 5 },
    { "Wilbur", 12 }
  };
  to_string(sc, ARRAY_SZ(sc), buffer, len);
}
```

Matthew Wilson <stlsoft@gmail.com>

This one is rather simple, I think: it's an issue of array->pointer decay.

The **ARRAY_SZ()** macro follows a familiar pattern in many codebases, used to fill the obvious missing **dimensionof()** operator that should

exist in C (and C++). It works by creating a compile-time constant representing the number of elements in an array by dividing the total size of the array by the size of an element. Works fine for arrays.

The problem is, it also works for pointers and, in C++, for types defining the subscript operator. And in such cases the sizes obtained is almost never correct (and sometimes not compile-time constant). This is all discussed at length in chapter 14 – Arrays and Pointers – of my book *Imperfect C++* [1], so I won't belabour the point here.

Also discussed in the same chapter of IC++ is the phenomenon of array-pointer decay. Briefly, it allows an array to be used in circumstances where a pointer is required. Also, a function declaration involving an array is interpreted, in a similar vein, as a pointer. Hence, the declaration of **process()** as

```
void process(char buffer[])
```

is exactly equivalent to the declaration

```
void process(char* buffer)
```

Expressed in this form, the problem is all too easy to see. The size of **ARRAY_SZ(buffer)** is going to be 4 (32-bit) or 8 (64-bit), and certainly not the 100 of the actual buffer **buffer** declared in **main()**.

The obvious fix is to change **process()** to have the signature

```
void process(char* buffer, size_t len)
```

which is hinted at strongly in the earlier defined **to_string()**.

When so amended – along with some const-correction, VC++ compatibility, and use of STLSoft's **STLSOFT_NUM_ELEMENTS()** (which makes application of **ARRAY_SZ()** to a pointer a compile-time, rather than runtime, error) – the code looks like:

```
#include <stlsoft/stlsoft.h>
#include <stdio.h>

#if defined(_MSC_VER)
# define snprintf _snprintf
#endif
#define ARRAY_SZ(x) STLSOFT_NUM_ELEMENTS(x)
typedef struct _Score
{
  char const* name;
  int         score;
} Score;
void to_string(Score const* scores, size_t n,
               char *buffer, size_t len)
{
  for (size_t i = 0; i < n; i++)
  {
    size_t printed = snprintf(buffer, len,
      "%s:\t%u\n",
      scores[i].name, scores[i].score);
    buffer += printed;
    len -= printed;
  }
}
void process(char* buffer, size_t len)
{
  Score const sc[] = {
    { "Roger", 10 },
    { "Bill", 5 },
    { "Wilbur", 12 },
  };
  to_string(sc, ARRAY_SZ(sc),
    buffer, len);
}
int main()
{
  char buffer[100];
  process(buffer, ARRAY_SZ(buffer));
  printf(buffer);
}
```

Reference

[1] *Imperfect C++*, Matthew Wilson, Addison-Wesley, 2004.

James Holland <James.Holland@babcockinternational.com>

My compiler did not provide any helpful hints as the student's program compiles without any errors or warnings. When I ran the program, "Rog" was displayed as the student observed. After a little investigation, I find that the problem is in the parameter of **process**. I suspect than the student is under the impression that an array is being passed to **process**, after all the type of the parameter, **char buffer[]**, looks like an array declaration. Unfortunately, despite its looks, the parameter type is equivalent to **char * buffer**. When the latter declaration is used, it becomes clear that a pointer to **buffer** is being passed to **process** and not **buffer** itself. From within **process**, it is the size of the pointer that is passed to **to_string** and not the size of **buffer**. On my machine, pointers are 4 bytes in length and so the value 4 is being passed to **to_string**. This explains why the program outputs "Rog". The function **to_string**, and ultimately **snprintf**, thinks that **buffer** is only 4 characters long, enough for three characters and the null terminator.

Unfortunately, it is not possible to pass to a function an array by value. Only a pointer to an array can be passed. This presents no great difficulty, however. If in addition to passing a pointer to the array, the length of the array is passed, **process** will have all the information it needs about **buffer**. Within **process**, the length of the output buffer can be passed straight to **to_string** as shown below.

```
void process(char * buffer, size_t length)
{
  Score sc[] = {{"Roger", 10}, {"Bill", 5},
    {"Wilbur", 12},};
  to_string(sc, ARRAY_SZ(sc), buffer, length);
}
```

When this modification is made, things look a lot better and the program produces the desired result. There are, however, some unresolved problems than could manifest themselves in the student's real program.

It is assumed that the student is using **snprintf** (as opposed to **sprintf**) because he/she does not want data to be written beyond the limits of **buffer**. This is a laudable desire but unfortunately the code, as it stands, does not provide that protection in general. There is still a potential problem when the size of the data to be written exceeds the size of **buffer**. This is demonstrated more conveniently if the size of **buffer** is reduced instead of increasing the size of the data.

Suppose, for example, that instead of **buffer** being declared 100 characters long, it had been declared 17 characters long. The program would correctly write to **buffer** the string representing the data for Roger. **snprintf** returns the length of the string (not including the null terminator) that it attempts to write, in this case, 10 characters. This value is assigned to the variable **printed**. This value is then subtracted from **len** (that currently has a value of 17) leaving 7. The program then attempts to write the data for Bill. This string is 8 characters long. The program (or more specifically **snprintf**) writes as much of the data to **buffer** as it can without overflowing **buffer**. Although the output string has been truncated, nothing disastrous (as far as program execution is concerned) has occurred. Next, the length of string for Bill is subtracted from **len**. The value of **len** is currently 7 and the length of the string for Bill is 8 characters. The subtraction to be performed is, therefore, 7 minus 8. This is where problems start. The variables **printed** and **len** are both of type **size_t** (an unsigned type). The result of the subtraction is not -1, as expected, but a very large positive number. On my machine the value of **len** at this point is 4294967295. The program then goes around the loop once more to write the data for Wilbur and, thinking there is plenty of space in **buffer** (because **len** is a large number), writes Wilbur's string beyond the end of **buffer**, probably with disastrous consequences.

This behaviour can easily be prevented by inserting the following code just after the **snprintf** statement. It will prevent the program going around the loop again when buffer is full and will also print a message explaining the problem.

```
if (printed >= len)
{
  printf("Buffer exhausted. Results may be "
    "incorrect.\n");
  break;
}
```

While on the subject of buffer length, there is another problem that occurs if **buffer** is declared as having zero size. I admit this is unlikely to occur in an otherwise fully debugged program but there is, at least, a theoretical point to be made here. An array of zero bytes will have an address, so it can be referenced. Such an array will, in all probability and not unreasonably, occupy zero bytes of memory. Given this, **printf** (as used in the last statement of **main**) will start printing at a memory location that has nothing to do with buffer. It could print anything of any length, depending on what data it stumbles across. This behaviour must be prevented if there is any possibility of **buffer** being of zero length. A simple **if** statement around **printf** would suffice. I leave the details to the student.

There is another problem with the **printf** statement; it is being used in a potentially unsafe way. **printf** is declared (in `stdio.h`) as having one or more parameters. The first parameter is a format control string. The student's **printf** statement has one parameter only and so the parameter must be the control string. It can be seen from an inspection of the **printf** statement that the control string is the contents of **buffer**. The contents of **buffer** came originally from the array **sc**. If one of the names in **sc** were to be changed, say, from "Roger" to "%sRoger", the program will probably crash in spectacular fashion. This is because **printf**'s format control string now says there is a string parameter to be printed despite there not actually being one. The result is undefined behaviour. To prevent this, a literal string should be supplied as the format control string and the string to be printed (in this case **buffer**) supplied as the second parameter as shown below.

```
printf("%s", buffer);
```

printf will now simply print the contents of **buffer** without interpreting any characters sequence as format control characters, as required.

Although it does not show itself in the student's test program, there is another potential problem lurking in the wings. It is the definition of the macro **ARRAY_SZ**. Suppose the student decides to make use of this macro somewhere else in his or her code. The student may write something like the following statement, for whatever reason.

```
size_t x = 25 % ARRAY_SZ(sc);
```

Let us assume that **sc** is the same array as defined in the student's test program. **ARRAY_SZ(sc)** should, therefore, produce the value 3. The expression becomes 25 % 3 which is equal to 1. The variable **x** should, therefore, be initialised with the value of 1. In fact **x** is assigned a value of 0, contrary to all expectation. How can this be? All becomes clear if the macro is expanded to produce the expression seen by the compiler, as illustrated below.

```
size_t x = 25 % sizeof(sc) / sizeof(sc[0]);
```

The expression will be evaluated from left to right. So the first term that is evaluated is **25 % sizeof(sc)** or 25 % 24 which equals 1. Next, the term **sizeof(sc[0])** is evaluated, this equals 8. Finally, 8 is divided into 1 giving zero. It can now be seen that brackets are required around **sizeof(sc) / sizeof(sc[0])** so that this part of the expression is evaluated first. This is achieved by enclosing the definition of **ARRAY_SZ(x)** in parentheses.

```
#define ARRAY_SZ(x) (sizeof(x)/sizeof(x[0]))
```

In fact it is good practice to enclose the definition of all but the simplest macros in parentheses to prevent this type of error.

There are a couple of inconsistencies in the program. **Score::score** is of type **int** and yet it is being printed by **snprintf** as if it were unsigned. Assuming **Score::score** is meant to be signed, then **snprintf**'s format control string should be **"%s:\t%d\n"**.

Also, quoted text strings are considered constant and yet **Score::name** is declared as a pointer that can modify what it is pointing to. The declaration **const char * name** provides the remedy. This will prevent accidentally attempting to write to a constant string as a compile-time error will result.

I noticed that the student provided a tag name when defining the **Scores** structure by use of a **typedef** statement. A structure need only have a tag name when the structure makes reference to pointers of the same type. This is not the case in the student's program and so the tag name is not required. Incidentally, it looks as if the student has made an attempt to prevent the tag name from clashing with the type name by use of a leading underscore character. This is not necessary as the two names are in different 'name spaces'. The tag and the type could both be named **Score**.

There is a redundant comma at the end of **SC**'s initialiser list. The compiler is perfectly happy with this and it does not affect the meaning of the program. It is allowed, at least in part, to make the job of automatic code generators a little easier. However, as it is not required, I prefer not to see a trailing comma in hand-written code.

Finally, it might be constructive to discuss the choice of variable names. To some degree this is a matter of personal style but there are some guidelines that should be observed. The names of variables (and other identifiers) should reflect their meaning and should not be excessively abbreviated. I would prefer to see **length_of_buffer** rather than **len** and **expected_print_length** rather than **printed**, for example. Selecting suitable names can be quite tricky and I am not suggesting my examples are ideal but I do think they help in understanding how the program works.

I have made quite a few corrections and suggestions in reviewing the student's code. This should not be seen as a damning criticism designed to dishearten the student. On the contrary, it is designed to encourage the student to complete his or her project in particular and to continue to learn about the fascinating topic of computer programming and software development in general.

Commentary

With five pretty comprehensive entries I'm not sure there's much for me to add. About the only thing no-one remarked on was the inconsistent brace positioning!

It seems a shame that **snprintf()** doesn't provide a foolproof safe replacement for **sprintf()** – examples like this code show how easy it is, given the variety of return values and the dangers of implicit conversion between signed and unsigned integer values, to use **snprintf()** in an unsafe way.

One additional point that might be worth making is that Visual Studio 2015 does provide **snprintf()** in `<stdio.h>` (although I haven't yet found where this is documented on MSDN.) This function seems to follow the behaviour required by the C11 standard. However, the Microsoft specific function **_snprintf()** is subtly *different*; in particular it does not ensure the target buffer is null terminated. This is an additional source of potentially dangerous confusion around this function call. Even beyond that, the implementation of **snprintf()** in the mingw implementation of g++ 4.9.2 on Windows *also* fails to ensure the buffer is null terminated.

The moral of this critique is to be extremely careful if you use **snprintf()** in C code – or in C++.

The Winner of CC 95

There were five good critiques and I think each one would have helped the person with the original problem to understand what was wrong and to fix it. However, I think Peter's critique covered the most ground and I have awarded him the prize for this issue's critique.

Code Critique 96

(Submissions to scc@accu.org by Oct 1st)

Thanks are due to Hubert Matthews for the idea that inspired this critique.

I have written some code to read in a CSV file and handle quoted strings but I seem to get an extra row read at the end, not sure why.

If I make a file consisting of one line:

--- Book1.csv ---

word,a simple phrase,"we can, if we want, embed commas"

--- ends ---

I get this output from processing the file:

```
Rows: 2
Cells: 3
  word
  a simple phrase
  "we can, if we want, embed commas"
Cells: 1
```

What have I done wrong?

The code is in Listing 2.

You can also get the current problem from the accu-general mail list (next entry is posted around the last issue's deadline) or from the ACCU website (http://www.accu.org/journals/). This particularly helps overseas members who typically get the magazine much later than members in the UK and Europe.

Listing 2

```
// Reading CSV with quoted strings.
#include <iostream>
#include <string>
#include <vector>

typedef std::string cell;
typedef std::vector<cell> row;
typedef std::vector<row> table;

table readTable()
{
  char ch;
  table table; // the table
  row *row = 0; // the row
  cell *cell = 0; // the cell
  char quoting = '\0';
  while (!std::cin.eof())
  {
    char ch = std::cin.get();
    switch (ch)
    {
    case '\n':
    case ',':
      if (!quoting) {
        cell = 0;
        if (ch == '\n') {
          row = 0;
        }
        break;
    case '\'':
    case '"':
        if (quoting == ch) {
          quoting = '\0';
        }
        else if (!cell) {
          quoting = ch;
        }
      }
    default:
      if (!row) {
        table.push_back({});
        row = &table.back();
      }
      if (!cell) {
        row->push_back({});
        cell = &row->back();
      }
      cell->push_back(ch);
      break;
    }
  }
  return table;
}

int main()
{
  table t = readTable();

  std::cout << "Rows: " << t.size() << "\n";
  for (int r = 0; r != t.size(); ++r) {
    std::cout << "Cells: "
      << t[r].size() << "\n";
    for (int c = 0; c != t[r].size(); ++c)
    {
      std::cout << "  " << t[r][c] << "\n";
    }
  }
}
```

Listing 2 (cont'd)

Write for us!

C Vu and Overload rely on article contributions from members. That's you! Without articles there are no magazines. We need articles at all levels of software development experience; you don't have to write about rocket science or brain surgery.

What do you have to contribute?

- What are you doing right now?
- What technology are you using?
- What did you just explain to someone?
- What techniques and idioms are you using?

For further information, contact the editors: cvu@accu.org or overload@accu.org

Robert Martin: An Interview

Emyr Williams continues the series of interviews with people from the world of programming.

Robert Martin (known as Uncle Bob) is a household name in computing. He's one of the original signatories of the Agile Manifesto (www.agilemanifesto.org) and also has a video blog that can be found here (link). He's been developing software for over forty years, and regularly speaks at international conferences. I happened to bump in to him at the ACCU conference last year, and he was kind enough to agree to be interviewed.

When did your interest in computing first start? Was it a sudden interest, or did it grow over time?

> It was very sudden. My mother bought a plastic computer for my 12th birthday. Digi-Comp-I. It was a mechanical three-bit finite state machine with three flip-flops and six and gates. The machine fascinated me. It was the first machine I ever programmed.

I was at the ACCU conference this year in April, and you started your lightning talk with a quick discourse on water, and what made it and so on. I am curious as to why you started your talk like that? Is it something you've always done?

> Twenty years ago, or so, I was teaching C++ quite frequently. Getting people to sit down and stop talking after breaks was difficult. So I started talking about science for 5 minutes. I found everyone stopped talking immediately and listened. Since then it's become a trade-mark of mine.

What was the first computer program that you ever wrote? Which language did you use to write it in?

> The program was called: "Mr. Patterson's Computerized gate." It was an instance of a three bit finite state machine that ran on my Digi-Comp-I computer. I was very proud of it at age 12.

Programming has come such a long way over the last few decades; programming languages and techniques have changed along with it - what would you say are the best things that have happened to programming; and conversely, what do you think are the worst?

> Actually, I disagree with the premise. Very little has happened in software over the last 40 years. The code written today is roughly the same as the code written 40 years ago. If statements, while loops, and assignment statements.
>
> All the major paradigms of software, Structured, OO, and Functional, were invented between the years 1957 and 1968. There has been very little new added since then.
>
> Our hardware has advanced miraculously. But our software has changed very little. A programmer from 1970 could read and write the code of today without much help. And if you took a modern programmer and transported him back to 1970, he'd be able to write the code without a lot of coaching.

I follow you on Twitter; fairly recently, there was talk of Test Driven Development being dead and that there seems to be a shift towards behaviour driven development. Would you say that TDD's days are numbered?

> I do not. I think TDD is still a growing practice. More and more people are adopting it, in spite of the "dead" meme. I believe TDD will eventually become a practice as important and universal to software developers and hand washing is to surgeons.

What would you say is the best piece of advice you've ever been given as a programmer?

> I wish I'd had someone to advise me in my earlier years. I had to learn what not to do by myself. If I could give advice to those who follow me, it would be: "The only way to go fast, is to go well. Go well in small steps."

Have you ever had a Eureka moment when you're coding or debugging? Could you tell us a little bit more about it?

> Eureka moments are more common in the shower, or driving home from work. There have been many, over the years. There have been times that I was half-way home from work and had to turn around to try my "Eureka" idea.

Did you have a mentor when you started programming? How did they make a difference to how you wrote your computer programs?

> I wish I'd had a mentor. What I had were books. Lots of books. At first they were the programming language manuals like Daniel D. McCracken's Fortran IV manual, and Kernighan and Ritchie's "The C Programming Language". Later I read books by Yourdon, Constantine, Demarco, Plauger, Booch, etc. Then, of course, came the books by Beck, Fowler, et. al.

Based on your experience, what would you say divides the truly great programmers from the average programmers?

> Great programmers take their time, and do much with little. Average programmers rush and do little with much.

A lot is said about elegant code today, and indeed you wrote a book on clean coding - what would you regard as the most elegant code you've seen?

> JUnit would be a candidate. So would the malloc/free implementation in Kernighan and Ritchie. But, above all those is the code in "The Structure and Interpretation of Computer Programs" by Abelson and Sussman. That stuff blew my mind.

If you could go back in time and meet yourself when you started, what would you tell yourself?

> Slow down. Don't rush. Get help often.

Finally, what advice would you give to anyone whether adult or kid, who's looking to start computer programming?

> It's a passion. Don't do it unless you love it. When I was a young programmer my friends and I would say to each other: "It's a good thing they pay us to write code; otherwise we'd have to pay them." And we would have. That's the kind of love of the art you need.

If you read something in C Vu that you particularly enjoyed, you disagreed with or that has just made you think, why not put pen to paper (or finger to keyboard) and tell us about it?

EMYR WILLIAMS

Emyr Williams is a C++ developer who is on a mission to become a better programmer. His blog can be found at www.becomingbetter.co.uk

View from the Chair

Alan Lenton
chair@accu.org

The *CVu* publication date comes round again. It seems like only yesterday that I was writing the piece for the last issue.

In that piece I mentioned that the Call for Papers for the 2016 ACCU Conference would be out in October. Well it's out now and you can find it at http://accu.org/index.php/conferences/accu_conference_2016/accu2016_call_for_sessions.

I really would encourage readers to consider presenting a paper at the conference. I know for sure that there is a wealth of experience in the membership, it's just struggling to get out and be shared. Both 90 minute and 15 minute slots are available, and, of course, there are also the highly entertaining and enlightening five minute 'lightning talks'. The deadline for the submission of proposals is Friday 13th November – an auspicious date, perhaps!

Confirmed keynote speakers are Andrei Alexandrescu and Jim Coplien (aka 'Cope'). The remaining two keynote speakers will probably have been announced by time your copy of *CVu* arrives. The conference runs from Wednesday 20th April to Saturday 23rd April, inclusive, and there are pre-conference tutorials and workshops the day before Conference starts, Tuesday 19th April. Get those dates into your diary for next year now, before you forget...

On a different note, we are starting to get occasional requests from recruiters for confirmation of candidates' ACCU membership. Now, we are, of course, happy that members put down their membership of ACCU on their CVs. However, whether we are authorised to confirm it, and possibly give information about how long you've been a member, to a third party, is something of a grey area.

So, to clarify things, if you do know that a confirmation is going to be required, please drop a line to the membership secretary, Matthew Jones, email: accumembership@accu.org, confirming that we have your permission to provide the information. If we haven't already got your permission, we will try to drop you a line requesting it. Your email info is up to date, of course. Isn't it?

Oh! And incidentally, do make sure your membership hasn't expired, and won't be expiring during lifetime of your CV. It's really embarrassing to have to tell the recruiter that the candidate is, in fact, no longer a member...

Finally, something that might be of interest to members. August 13th this year was the bicentenary of the birth of Ada Lovelace, and to celebrate it, the Science Museum has a free exhibition about her. The exhibition brings together portraits, letters and notes, as well as Babbage's Difference and Analytic Engines. If you live in London, or you're visiting, it looks like it might well be worth seeing (see http://www.sciencemuseum.org.uk/visitmuseum/Plan_your_visit/exhibitions/ada-lovelace). I certainly intend to take a look, so perhaps I'll see you there. The exhibition runs until 31st March 2016.

Well, that's about all there is to tell you about for this issue, since it's fairly quiet at the moment... In the meantime, I look forward to seeing the proposals for conference sessions flooding in!

Enjoy your programming,

Standards Report

Jonathan Wakeley reports on developments in C++.

I'm writing this report at Heathrow airport, about to fly out to Kona, Hawaii, for the C++ standards meeting. By the time you read this the C++ committee and the C committee will both have finished their Kona meetings (which run back to back) so expect to see news from those meetings in my next report. Although going to Kona sounds exciting, the C++ committee have well over 100 papers to discuss during the week and I don't expect to see much outside the meeting rooms. I wonder if the C committee have a more relaxed schedule and can enjoy the nice location, and if I'm on the wrong committee!

Since my last report for *CVu* there haven't been any face-to-face meetings, but the pre-meeting mailing [1] for the C++ meeting has lots of papers. The sharp-eyed will notice a change in the naming scheme for papers. From now on N-numbers will only be used for official ISO documents such as meeting notices, agendas and minutes, and working drafts and project editors' reports. The informal proposals and position papers will get a P-number, with a suffix indicating whether it is a revision of an earlier paper.

The mailing includes a new working draft for 'Ranges' [2], which I mentioned in my last report [3]. The lengthy email discussions on the `std::experimental::variant` design have abated, but there are several papers in the mailing about variant and it will be a hot topic during the Kona meeting. At the end of the Kona meeting the Networking proposal [4] should be in good shape to turn it into a working draft, which is the next step towards publishing a TS. There is a proposal to add the content of the Parallelism TS to C++17, which will no doubt be discussed in Kona too.

Apart from the upcoming meeting, the other big C++ news is the announcement at CppCon of the 'C++ Core Guidelines' [5] which are a set of (still evolving) coding guidelines intended to encourage people to use the modern, safe features of the language. Part of that also requires discouraging people from using the dark corners of the language which are still part of the standard but have no place in most code. Accompanying the guidelines will be an open-source library providing a set of useful vocabulary types and utilities to help follow the guidelines, and a static analysis tool to check that code conforms to the guidelines. Keep your eyes peeled for these soon.

References

[1] http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2015/#mailing2015-09
[2] http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2015/p0021r0.pdf
[3] *CVu* 27-4, September 2015
[4] http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2015/p0112r0.html
[5] https://isocpp.org/blog/2015/09/bjarne-stroustrup-announces-cpp-core-guidelines

JONATHAN WAKELY

Jonathan's interest in C++ and free software began at university and led to working in the tools team at Red Hat, via the market research and financial sectors. He works on GCC's C++ Standard Library and participates in the C++ standards committee. He can be reached at accu@kayari.org